Syllabus

Unit	Unit Outcomes (UOs) (in cognitive domain)	Topics and Sub-topics
Unit - I : 8086- 16 Bit Microprocessor Refer chapter 1	1a. Describe function of the given pin of 8086. 1b. Explain with sketches the working of given unit in 8086 microprocessor. 1c. State functions of the given registers of 8086 microprocessor. 1d. Calculate the physical address for the given segmentation of 8086 microprocessor.	1.1 8086 Microprocessor : Salient features, Pin descriptions 1.2 Architecture of 8086 : Functional Block diagram, Register organization 1.3 Concepts of pipelining 1.4 Memory segmentation, Physical memory addresses generation.
Unit - II : The Art of Assembly Language Programming Refer chapter 2	2a. Describe the given steps of program development / execution. 2b. Write steps to develop a code for the given problem using assembly language programming. 2c. Use relevant command of debugger to correct the specified programming error. 2d. Describe function of the given assembler directives with example.	2.1 Program development steps : Defining problem and constrains, Writing Algorithms, Flowchart, Initialization checklist, Choosing instructions, Converting algorithms to assembly language programs 2.2 Assembly Language Programming Tools : Editors, Assembler, Linker, Debugger 2.3 Assembler directives.
Unit - III : Instruction Set of 8086 Microprocessor Refer chapter 3	3a. Determine the length of the given instruction. 3b. Describe the given addressing modes with examples. 3c. Explain the operation performed by the given instruction during its execution. 3d. Identify the addressing modes in the given instructions.	3.1 Machine Language Instruction format 3.2 Addressing modes 3.3 Instruction set, Groups of Instructions : Arithmetic instructions, Logical Instructions, Data transfer instructions, Bit manipulation instructions, String Operation instructions, Program control transfer or branching instructions, Process control instructions.
Unit - IV : Assembly Language Programming Refer chapter 4	4a. Use the given model of assembly language programs for the given problem. 4b. Develop the relevant program for the given problem. 4c. Apply relevant control loops in the program for the given problem. 4d. Use string instructions for the given strings/block to manipulate its elements.	4.1 Model of 8086 assembly language programs 4.2 Programming using assembler : Arithmetic operations on Hex and BCD numbers, Sum of Series, Smallest and Largest numbers from array, Sorting numbers in Ascending and Descending order, Finding ODD, EVEN, Positive and Negative numbers in the array, Block transfer, String Operations - Length, Reverse, Compare, Concatenation, Copy, Count Numbers of '1' and '0' in 16 bit number.
Unit - V : Procedure and Macro Refer chapter 5	5a. Apply the relevant 'parameter-passing' method in the given situation. 5b. Develop an assembly language program using the relevant procedure for the given problem. 5c. Develop an assembly language program using MACROS for the given problem. 5d. Compare procedures and macros on the basis of the given parameters.	5.1 Procedure : Defining and calling Procedure - PROC, ENDP, FAR and NEAR Directives; CALL and RET instructions; Parameter passing methods, Assembly Language Programs using Procedure 5.2 Macro : Defining Macros, MACRO and ENDM Directives, Macro with parameters, Assembly Language Programs using Macros.

List of Practicals

Sr. No.	Practical Outcomes (POs)	Unit no.	Approx. Hrs. Required
1	Identify various pins of the 8086 Microprocessor.	I	02*
2	Use assembly language programming tools and functions.	II	02*
3	Use different addressing mode instruction in program (a) Write an Assembly Language Program (ALP) to add two given 8 and 16 bit numbers. (b) Write an Assembly Language Program (ALP) to subtract two given 8 and 16 bit numbers.	III	02*
4	(a) Write an ALP to multiply two given 8 and 16 bit unsigned numbers. (b) Write an ALP to multiply two given 8 and 16 bit signed numbers.	III	02
5	(a) Write an ALP to perform block transfer data using string instructions. (b) Write an ALP to perform block transfer data without using string instructions.	III	02
6	(a) Write an ALP to compare two strings without using string instructions. (b) Write an ALP to compare two strings using string instructions.	III	02
7	(a) Write an ALP to divide two unsigned numbers. (b) Write an ALP to divide two signed numbers.	III	02
8	Write an ALP to add, subtract, multiply and divide two BCD numbers.	IV	02
9	Implement loop in assembly language program (a) Write an ALP to find sum of series of hexadecimal numbers. (b) Write an ALP to find sum of series of BCD numbers.	IV	02*
10	(a) Write an ALP to find smallest number from array of n numbers. (b) Write an ALP to find largest number from the array of n numbers.	IV	02*
11	(a) Write an ALP to arrange numbers in array in ascending order. (b) Write an ALP to arrange numbers in array in descending order.	IV	02
12	(a) Write an ALP to arrange string in reverse order. (b) Write an ALP to find string length. (c) Write an ALP to concatenation of two strings.	IV	02
13	(a) Write an ALP to check given 16-bit number is odd or even. (b) Write an ALP to count ODD and/or EVEN numbers from the array of five 16-bit numbers.	IV	02
14	(a) Write an ALP to check given number is positive or negative. (b) Write an ALP to count Positive and/or Negative numbers in an array.	IV	02
15	(a) Write an ALP to count numbers of '1' in a given number. (b) Write an ALP to count numbers of '0' in a given number.	IV	02
16	An assembly language program using procedures (a) Write an ALP addition, subtraction, multiplication and division. (b) Write an ALP for using procedure to solve equation such as Z = (A + B) * (C + D).	V	02*
17	Write an assembly language program using macros (a) Write an ALP for addition, subtraction, multiplication and division. (b) Write an ALP using MACRO to solve equation such as Z = (A + B) * (C + D).	V	02*
	Total		**34**

Note : The practicals marked as '*' are compulsory.

❑❑❑

UNIT - I

Syllabus : 8086 Microprocessor : Salient features, Pin descriptions, Architecture of 8086 - Functional Block diagram, Register organization, Concepts of pipelining, Memory segmentation, Physical memory addresses generation.

UNIT - II

Syllabus : Program development steps : Defining problem and Constrains, Writing Algorithms, Flowchart, Initialization checklist, Choosing instructions, Converting algorithms to assembly language programs.

Assembly Language Programming Tools : Editors, Assembler, Linker, Debugger, Assembler directives.

UNIT - III

Syllabus : Machine Language Instruction format, Addressing modes. Instruction set, Groups of Instructions : Arithmetic Instructions; Logical or Bit manipulation Instructions; Data transfer instructions; String Operation Instructions; Program control transfer or branching Instructions; Process (Machine) control Instructions.

UNIT - IV

Syllabus : Model of 8086 Assembly Language Programs. Programming Using assembler : Arithmetic operations on Hex and BCD numbers, Sum of series, Smallest and Largest numbers from array, Sorting numbers in Ascending and Descending order, Finding ODD, EVEN positive and negative numbers in the array, Block transfer String operations Length, Reverse, Compare, Concatenation, Copy, Count numbers of '1' and '0' in 16 bit number.

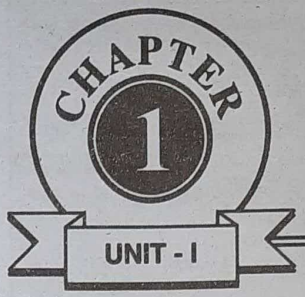
UNIT - V

Syllabus : Procedure : Defining and calling, Procedure - PROC, ENDP, FAR, and NEAR Directives; CALL and RET instructions; Parameter passing methods, Assembly Language, Programs using Procedure.

Macro : Defining Macros, MACRO and ENDM Directives. Macro with parameters. Assembly Language Programs using Macros.

List of Practicals

❑❑❑



8086-16 Bit Microprocessor

Syllabus

8086 Microprocessor : Salient features, Pin descriptions, Architecture of 8086 - Functional Block diagram, Register organization, Concepts of pipelining, Memory segmentation, Physical memory addresses generation.

Syllabus Topic : 8086 Microprocessor

1.1 Introduction

- The 8086 is the first 16-bit microprocessor developed in 1978 by an Intel Corporation.
- The 8086 microprocessor has a much more powerful instruction set along with the architecture developments which provides programming flexibility and improves the speed of operation as compare to 8-bit microprocessor.
- The 8086 is a 16-bit HMOS microprocessor. It is available in a 40 pin IC and operates at 5 volts DC supply.
- Its electronic circuitry consists of 29000 transistors. It is implemented in N-channel, silicon gate technology and available in three versions i.e. 8086(5 MHz), 8086-2(8 MHz) and 8086-1(10 MHz).
- The 8086 microprocessor is no longer used, but the concept of its principles and structure is very much useful for the understanding of other advanced Intel microprocessors.
- The 8086 have 20 address lines using which we can interface 2^{20} = 1 MB of memory mean it can address up to 1 MB memory.
- Out of 20 address lines, 16 address lines are multiplexed with data line and named as AD_0-AD_{15}.
- Remaining four address lines are also multiplexed with status signals.

Syllabus Topic : Salient Features

1.2 Salient Feature of 8086 Microprocessor

➔ **(MSBTE - S-14, W-16, S-17, W-17)**

Q. 1.2.1 List any four advantages of 8086 microprocessor. (Ref. sec. 1.2)

Q. 1.2.2 State any eight features of 8086. (Ref. sec. 1.2)

Q. 1.2.3 State the maximum size of memory which can be interface to 8086. Why? (Ref. sec. 1.2) S-14, 2 Marks

Q. 1.2.4 List any four features of 8086. (Ref. sec. 1.2) W-16, S- 17, W- 17, 2 Marks

- Provides 20 address lines, so 1024 Kbyte (1Mbytes) of memory can be addressed.
- Multiplexed 16-bit address and data bus AD_0 – AD_7 to minimize numbers of pin on IC.
- Operating clock frequencies are 5 MHz, 8 MHz, 10 MHz
- Arithmetic operation can be performed on 8 bit or 16 bit signed and unsigned data including multiplication and division.
- Operates in single processor and multiprocessor configuration i.e. operating modes.
- The instruction set is powerful, flexible and can be programmed in high level language like C language.

- Provides 256 types of vectored software interrupts.
- Provide 6-byte instruction queue for pipelining of instructions execution.
- Generate 8 bit or 16 bit I/O address so it can access maximum 64 K I/O devices.
- Operate in maximum and minimum mode to achieve high performance level.
- Supports 24 different addressing modes.
- Supports multiprogramming.
- Provides separate instructions for string manipulation.

Syllabus Topic : Pin Description

1.3 Pin Description of 8086

➔ **(MSBTE - S-14, W-14, S-15, W-15, S-16, W-16, S-17, W-17, S-18)**

Q. 1.3.1 Draw the pin diagram of 8086. (Ref. sec. 1.3) S-14, 4 Marks

Q. 1.3.2 State the function of following pins of 8086.

(a) $\overline{TEST}$ (b) $\overline{BHE}$

(c) $\overline{INTA}$ (d) $DT/\overline{R}$

(Ref. sec. 1.3) S-14, 4 Marks

Q. 1.3.3 State the function of following pins of 8086.

(i) NMI (ii) $\overline{TEST}$

(iii) $\overline{DEN}$ (iv) $\overline{MN}$

(Ref. sec. 1.3) S-15, S-16, 4 Marks

Q. 1.3.4 State all the control signals generated by S0, S1 and S2 with their functions. (Ref. sec. 1.3) W-14, S-16, W-17, S-18, 4 Marks

Q. 1.3.5 State the function of following pins of 8086 microprocessor.

(i) ALE (ii) $DT/\overline{R}$

(iii) $M/\overline{IO}$ (iv) HOLD

(Ref. sec. 1.3) W-14, W-15, 4 Marks

Q. 1.3.6 Explain maskable and non-maskable interrupt used in 8086. (Ref. sec. 1.3) S-15, 4 Marks

Q. 1.3.7 State the functions of following pins of 8086.

1) ALE 2) $\overline{WR}$

(Ref. sec. 1.3) W-16, 2 Marks

Q. 1.3.8 Explain the function of following pins of 8086 microprocessor.

(i) $MN/\overline{MX}$ (ii) READY

(iii) ALE (iv) $DT/\overline{R}$

(Ref. sec. 1.3) S-17, W-17, 4 Marks

Q. 1.3.9 State the function of following pins of 8086 microprocessor.

(i) $DT/\overline{R}$ (ii) NMI

(iii) $\overline{RD}$ (iv) $\overline{DEN}$

(Ref. sec. 1.3) S-18, 4 Marks

The Fig. 1.3.1 shows the pin diagram of 8086 microprocessor.

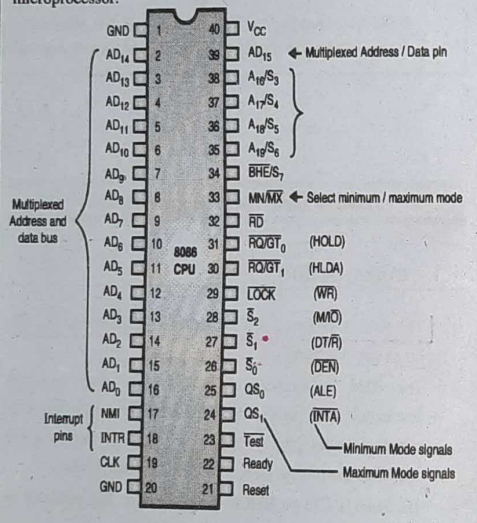


Fig. 1.3.1 : Pin diagram of 8086 Microprocessor

AD_0-AD_{15}

- These lines are time-multiplexed bi-directional tristate address/data bus.
- During T_1 clock cycle of the bus cycle, these lines carry lower order 16-bit address.
- During T_2, T_3 and T_4, they carry 16-bit data.
- So, AD_0-AD_7 lines carry lower order byte of data and AD_8-AD_{15} carry high order byte of data.

A_{19}/S_6, A_{18}/S_5, A_{17}/S_4, A_{16}/S_3

- These are time sharing multiplexed address and status lines.
- During T_1 clock cycle, these lines carry upper four-bit address and during I/O operation, these lines are low.

- During T_2, T_3 and T_4, S_3 and S_4 carry status signal and these status line are used to identify memory segments as shown in Table 1.3.1.
- S_5 is an interrupt enable status signal and is updated at the beginning of each clock cycle.

Table 1.3.1

S_4	S_3	Segment register
0	0	ES
0	1	SS
1	0	CS or none
1	1	DS

$\overline{BHE}$ / S_7 (Bus High Enable / Status)

- The Bus High Enable signal is used to indicate the transfer of data over higher order (D_{15}-D_8) data bus shown in Table 1.3.2.
- It goes low for the data transfer over D_8 - D_{15} and is used to drive chip selects of odd address memory bank or peripherals.
- $\overline{BHE}$ in combination with A_0 determines whether a byte or word will be transferred from / to memory locations.
- The $\overline{BHE}/S_7$ is a time-multiplexed line, so during T_2 to T_4 the status signal S_7 is transmitted on this line.
- It remains always high.

Table 1.3.2

$\overline{BHE}$	A_0	Word / Byte Access
0	0	Whole word from even address
0	1	Upper byte from/to odd address
1	0	Lower byte from/to even address
1	1	None

$\overline{RD}$ (Read)

- It is an active low read signal generated by the processor to indicate that the processor is performing read operation with memory or I/O depending on the status of $M/\overline{IO}$ signal.
- This signal is used to read devices, which are connected to the 8086 local bus and remain tri-stated during 'hold acknowledge' (HLDA).

READY

- This is an acknowledgement signal from the slower I/O device or memory.
- It is an active high input signal used to synchronize slower peripheral/memory with faster microprocessor.
- When high, it indicates that the peripheral device is ready to transfer data.

Q. 1.3.10 How is an 8086 entered into a wait state ? (Ref. sec. 1.3)

- READY pin can be used to add wait state. When this pin is high, the 8086 is "READY" and operates normally.
- If the READY input is made low at the right time in a machine cycle, the 8086 will insert one or more wait state between T_3 and T_4 in that machine cycle.
- An external hardware device is set up to pulse READY low before the rising edge of the clock in T_2.
- After the 8086 finishes T_3 of the machine cycle, it enters the wait state.
- During a WAIT state, the signals on the buses remaining the same as they at the start of the WAIT state.
- The address of the addressed memory location is held on the output of latches, so it does not change, the control bus signal $\overline{M/IO}$ and $\overline{RD}$, also do not change DURING the WAIT state, T_{WAIT}.
- The memory or port device then has at least one more clock cycle to gets its data output.
- If the READY input is made high again during T_3 or during the WAIT state, then after one WAIT state, the 8086 will go on with the regular T_4 of the machine cycle.
- If the 8086 READY input is still low at the end of a WAIT state, then the 8086 will insert another WAIT state.
- The 8086 will continue inserting WAIT states until the READY input is made high again.

RESET

- It is a system reset.
- When this signal goes high, processor enter into reset state and terminate the current activity and start execution from ***FFFF0 H***.
- This signal is an active high signal and must be active for at least four clock cycles.

Maskable and Non-maskable Interrupt

- Maskable hardware interrupts can be mask or unmask and the 8-bit vector type must be provided by an interrupting device to the processor during an interrupt acknowledge bus sequence.
- In 8086, INTR is an maskable interrupt.
- Non-maskable hardware interrupts use a predefined internally supplied vector and cannot be masked or avoided, processor has to service these interrupts.
- In 8086, NMI and all software interrupt are non maskable.

INTR (Interrupt Request)

- This is a level triggered interrupt request input and checked during the last clock cycle of each instruction to determine the availability of the request.
- If any interrupt request is occurred, the processor enters the interrupt acknowledge cycle.

NMI (Non-maskable Interrupt)

- This is an edge triggered input interrupt request which causes a Type-2 interrupt.
- The NMI is not maskable by software.

$\overline{TEST}$

- This signal is used to test the status of math co-processor 8087.
- The $\overline{BUSY}$ pin of 8087 is connected to this pin of 8086.
- This input signal is examined by a 'WAIT' instruction.
- If the $\overline{TEST}$ signal goes low, execution will continue, else, the processor remains in an idle state.

CLK (Clock input)

- This clock input provides the basic timing for processor operation and bus control activity.
- It is symmetric square wave with 33% duty cycle.
- The range of frequency for different 8086 versions is from 5 MHz to 10 MHz.
- V_{cc} - +5 V power supply for the operation of internal circuit.
- GND - Ground for the internal circuit.

MN / $\overline{MX}$

- This pin indicates the operating mode of 8086.
- There are two operating modes of 8086 i.e. minimum mode and maximum mode.
- When this pin is connected to V_{cc}, the processor operates in minimum mode and when this pin is connected to ground, processor operates in maximum mode.

For minimum mode of operation, the pin MN/$\overline{MX}$ is kept high i.e. connected to +5 V. The pins 24 - 31 have unique function for minimum mode of operation as given below.

Pin 24 : $\overline{INTA}$ (Interrupt acknowledge)

- It is an active low output signal interrupt acknowledge signal.
- When processor receive INTR signal, the processor complete current machine cycle and acknowledge the interrupt by generating this signal.

Pin 25 : ALE (Address latch enable)

- It is an active high pulse issued by the processor during T_1 state of bus cycle to indicate the availability of valid address on the AD_0-AD_{15}.
- This pin is connected to latch enable pin of latches 8282 or 74LS373.

Pin 26 : $\overline{DEN}$ (Data enable)

- It is an active low signal issued by the processor during middle of T_2 until the middle of T_4 to indicate the availability of valid data over AD_0-AD_{15}.
- This signal is used to enable the transceivers (bi-directional buffers) 8286 or 74LS245 to separate the data from the multiplexed address / data signals.

Pin 27 : DT/$\overline{R}$ (Data transmit / receive)

- This output signal is used to decide the direction of data flow through the transceivers (bi-directional buffers) 8286 / 74LS245.
- When the processor sends data out, this signal is high and when the processor receives data, then this signal is low.

Pin 28 : M/$\overline{IO}$ (status signal)

- This signal is issued by the processor to distinguish memory access from an I/O access.
- When this signal is high, memory is accessed and when this signal is low, an I/O device is accessed.

Pin 29 : $\overline{WR}$ (Write)

- It is an active low signal issued by the processor to write data to memory or I/O device depending on the status of M/$\overline{IO}$ signal.

Pin 30 : HLDA (Hold acknowledge)

- This is an active high output signal generated by the processor after receiving the HOLD signal.

Pin 31 : HOLD

- When another master device such as DMA Controller needs the use of the address, data and control bus, it sends a HOLD request to the processor through this line.
- It is an active high input signal.

For maximum mode of operation, the pin MN/$\overline{MX}$ is kept low. The pins 24 - 31 have unique function for maximum mode of operation as given below :

Pin 24, 25 : QS_0, QS_1 (Queue Status)

- These lines provide information about the status of instruction queue during the clock cycle after which the queue operation is performed.
- The Table 1.3.3 shows the status of QS_0 and QS_1.

Table 1.3.3

QS_1	QS_0	Status
0	0	No Operation
0	1	1st byte of op-code from queue
1	0	Empty queue
1	1	Subsequent byte from queue

Pin 26, 27, 28 : $\overline{S_0}$ $\overline{S_1}$ $\overline{S_2}$ (Status signal)

- These status signals show the type of operation, being carried out by the processor and required by the bus controller 8288 to generate all memory or I/O access control signals.
- These signals become active during T_4 of previous cycle and remain active during T_1 and T_2 of the current cycle.
- These status lines are encoded as given in Table 1.3.4.

Table 1.3.4

$\overline{S_2}$	$\overline{S_1}$	$\overline{S_0}$	Status
0	0	0	Interrupt acknowledge
0	0	1	I/O Read
0	1	0	I/O Write
0	1	1	Halt
1	0	0	Op-code Fetch
1	0	1	Memory Read
1	1	0	Memory Write
1	1	1	Passive

Pin 29 : $\overline{LOCK}$

- This is an active low output signal used to prevent other system bus master from gaining the system bus, while the $\overline{LOCK}$ signal is low and generated by LOCK prefix instruction.
- When it goes low, all interrupts are masked and HOLD request is not granted.

Pin 30, 31 : $\overline{RQ}$ / $\overline{GT_0}$, $\overline{RQ}$ / $\overline{GT_1}$ (Request / Grant)

- These pins are used by other local bus master such as DMA Controller in maximum mode to gain the control of local buses at the end of the processor's current bus cycle.
- The pins $\overline{RQ}$ / $\overline{GT_0}$ and $\overline{RQ}$ /$\overline{GT_1}$ are bi-directional and $\overline{RQ}$ / $\overline{GT_0}$ have higher priority than $\overline{RQ}$ /$\overline{GT_1}$.
- After receiving request on these lines, the CPU sends acknowledge signal on same lines.

Syllabus Topic : Architecture of 8086 : Functional Block Diagram, Register Organization

1.4 Internal Architecture of 8086

➔ **(MSBTE - W-14, S-15, W-15, S-16, W-16, S-17, W-17)**

Q. 1.4.1 Write any four important functions of any two units of 8086 microprocessor. (Ref. sec. 1.4) **W-14, 4 Marks**

Q. 1.4.2 Draw architecture of 8086 and label it. Write the function of BIU and EU. (Ref. secs. 1.4, 1.4.1 and 1.4.2) **S-15, 8 Marks**

Q. 1.4.3 List all 16 bit register in 8086 and write their instructions. (Ref. sec. 1.4) **S-15, 4 Marks**

Q. 1.4.4 Draw the architecture of 8086 microprocessor and state the function of BIU. (Ref. secs. 1.4 and 1.4.2) **W-15, 4 Marks**

Q. 1.4.5 Draw the neat labeled architecture diagram of 8086 microprocessor. (Ref. sec. 1.4) **S-16, 4 Marks**

Q. 1.4.6 Draw the functional block diagram of 8086 microprocessor and describe instruction queue in detail. (Ref. secs. 1.4 and 1.4.2) **W-16, 8 Marks**

Q. 1.4.7 State the function of following register of 8086 microprocessor.
(i) General purpose register
(ii) Segment register
(Ref. secs. 1.4.1 and 1.4.2) **S-17, 4 Marks**

Q. 1.4.8 Describe register organization of 8086. (Ref. sec. 1.4) **W-17, 4 Marks**

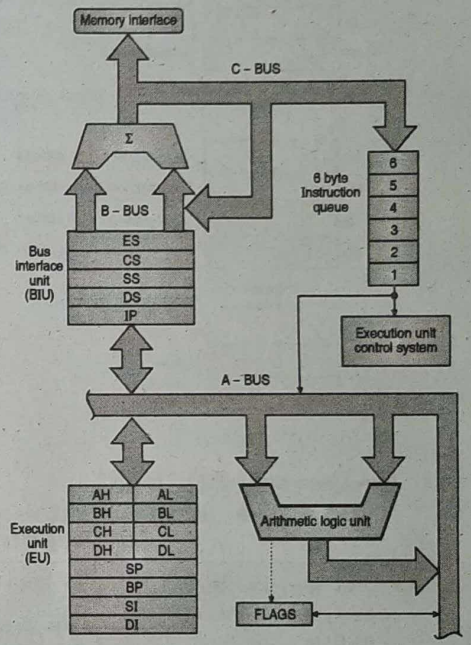


Fig. 1.4.1 : Architecture or functional block diagram of 8086

As shown in Fig. 1.4.1, the 8086 CPU is divided into two independent functional parts i.e.

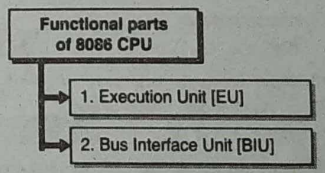


Fig. 1.4.2 : Functional parts of 8086 CPU

Register Structure with their functions

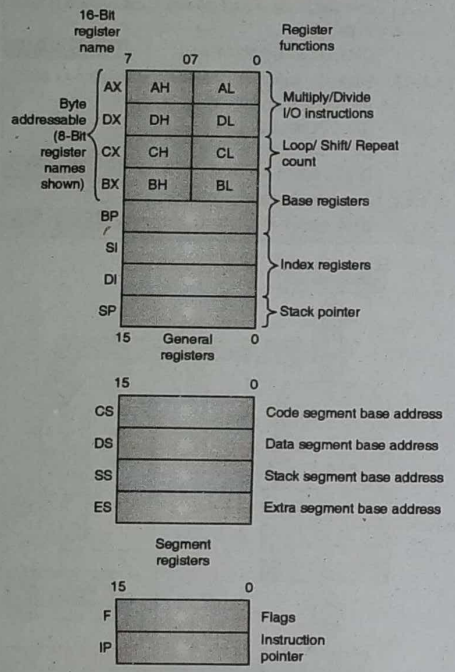


Fig. 1.4.3

1.4.1 Execution Unit [EU]

➔ **(MSBTE - S-14, W-14, W-15, S-16, W-16, S-18)**

Q. 1.4.9 Draw the flag register format of 8086 microprocessor and explain any two flags. (Ref. sec. 1.4.1) S-14, W-15, 4 Marks

Q. 1.4.10 Draw the neat labeled architecture of flag register of 8086 microprocessor. (Ref. sec. 1.4.1) W-14, 4 Marks

Q. 1.4.11 Draw labeled flag register format of 8086 microprocessor. (Ref. sec. 1.4.1) S-16, 2 Marks

Q. 1.4.12 Name the general purpose registers of 8086 giving brief description of each. (Ref. sec. 1.4.1) W-16, S-18, 4 Marks

Q. 1.4.13 State the use of OF, TF, AF and PF flags in 8086. (Ref. sec. 1.4.1) S-18, 4 Marks

- The functions of execution unit are :
  1. To tell BIU where to fetch the instructions or data from.
  2. To decode the instructions.
  3. To execute the instructions.
- The EU contains the control circuitry to perform various internal operations.
- A decoder in EU decodes the instruction fetched from memory to generate different internal or external control signals required to perform the operation.
- EU has 16-bit ALU, which can perform arithmetic and logical operations on 8-bit as well as 16-bit data.
- Flag register in EU is of 16-bit and shown in Fig. 1.4.5.
- These registers contain nine active flags.
- Five flags in the lower byte of this register are similar to 8085 flag register.

So, the flag register of 8086 is divided into two parts i.e. condition code or status flags and machine control flags.

- The condition code flag register including overflow flag of 8086 reflects the result of the operation performed by the ALU.
- The machine control flags are Direction Flag DF, Interrupt Flag IF and Trap Flag TF.

Status flags

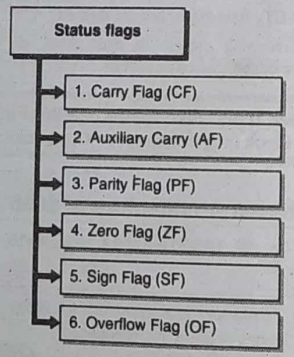


Fig. 1.4.4 : Status Flags

➔ 1. Carry Flag (CF)

- It is set to 1 if there is carry out of the MSB position i.e. resulting from an addition or if a barrow is needed at MSB during subtraction.
- If there is no carry/barrow out of MSB bit of result, the CF flag is reset.

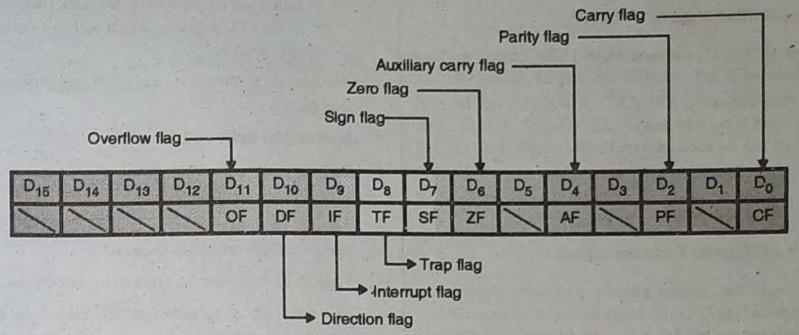


Fig. 1.4.5 : Flag register format

➔ 2. Auxiliary Carry (AF)

- If an operation performed in ALU generates a carry/barrow from lower nibble (i.e. D_0-D_3) to upper nibble (i.e. D_4-D_7), the AF flag is set i.e. carry given by D_3 bit to D_4 is AF flag.
- This is not general-purpose flag; it is used internally by the processor to perform binary to BCD conversion.

➔ 3. Parity Flag (PF)

- This flag is used to indicate the parity of result.
- If lower order 8-bits of the result of an operation contains even number if 1, the parity flag is set and for odd number of 1, the parity flag is reset.

➔ 4. Zero Flag (ZF)

- It is set, if the result of arithmetic or logical operation is zero else it will be reset.

➔ 5. Sign Flag (SF)

- In sign magnitude format the sign of number is indicated by MSB bit.
- If the result of operation is negative, sign flag is set.
- The sign flag is replica of MSB bit of result.

➔ 6. Overflow Flag (OF)

- In case of the signed arithmetic operation, the overflow flag is set, if the result is too large to fit in the numbers bits available to accommodate it.
- The overflow flag has no significance in unsigned arithmetic operation.

Difference between Carry and Overflow Flags

Carry Flag	Overflow Flag
Generated during the arithmetic and logical operation on unsigned numbers	Generated during the arithmetic and logical operation on signed numbers
Generated by D7 or D15 bit of 8 or 16 bit number	Generated by D6 or D14 bit of 8 or 16 bit number

The three control flags

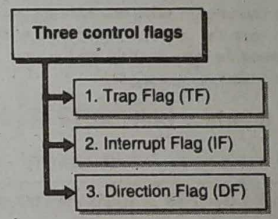


Fig. 1.4.6 : Three control flags

➔ 1. Trap Flag (TF)

- It is used for single step control.
- It allows user to execute one instruction of a program at a time for debugging.
- When trap flag is set, the program can be run in single step mode.

➔ 2. Interrupt Flag (IF)

- It is an interrupt enable / disable flag.
- If it is set, the maskable interrupt INTR of 8086 is enabled and if it is reset, the interrupt is disabled.
- It can be set by executing instruction STI and can be cleared by executing CLI instruction.

➔ 3. Direction Flag (DF)

- The DF (Direction flag) is used in string operation.
- If DF is set, string bytes are read or write from higher memory address to lower memory address.
- If DF is reset, the string bytes are read or write from lower memory address to higher memory address.
- The DF can be set by executing STD instruction and can be reset by executing CLD instruction.

General purpose registers of 8086

- Execution Unit (EU) contains eight 16-bit general purpose registers named as AX, BX, CX, DX, SP, BP, SI and DI.
- Out of these registers, AX, BX, CX and DX can be used either as eight 8-bit registers i.e. AL, AH, BL, BH, CL, CH, DL, DH or can be used as four 16-bit i.e. AX, BX, CX and DX.
- The AL register is called as 8-bit accumulator and AX is called as 16-bit accumulator.

Function of General Purpose register

- In addition to the general purpose job, some register has special task such as CX is normally used as a counter, BX can be used as a pointer and DX is used for I/O addressing to hold the I/O address in some instructions of the 8086 microprocessor.
- The other registers in EU are SP, BP, SI and DI. SP and BP are pointer register, which holds 16-bit offset within the particular segment. SI and DI are the index registers.
- During the execution of string related instructions, register SI is used to store the offset of source data or string in data segment while the register DI is used to store the offset of destination in data or extra segment.

1.4.2 Bus Interface Unit [BIU]

→ **(MSBTE - W-14, S-18)**

Q. 1.4.14 State the names of segment registers in 8086 microprocessor. (Ref. sec. 1.4.2) W-14, S-18, 4 Marks

The function of BIU is to send address to:

(a) Fetch the instruction or data from memory.
(b) Write the data to memory.
(c) Write the data to the port.
(d) Read data from the port.

Various sections of the BIU are given below.

Segment Registers

- BIU has 4 segment registers of 16-bit each i.e. CS, DS, SS and ES.
- The memory pointers are used to point or address particular memory location in memory
- Following register acts as the memory pointers register in 8086.
  - The Code Segment CS register is used to address a memory location in the code segment of the memory, where the op-code of program is stored.
  - The Data Segment DS register points to the data segment of the memory, where the date is stored.
  - The Extra Segment ES register is used to address the segment which is additional data segment used to store data.
  - The Stack Segment SS register is used to points stack location in stack segment of the memory and used to store data temporarily on the stack such as the contents of the CPU registers, which will be required later stage of execution.
- *The Default Segment base and offset pair registers are CS: IP and SS: SP.*

Instruction queue IQ (Queue)

- To increase the execution speed, BIU fetches as many as six instruction bytes ahead to time from memory.
- All the six bytes are then held in first-in-first-out 6-byte register called instruction queue IQ.
- Then all bytes have to be given to EU one-by-one.
- This pre-fetching operation of BIU may be in parallel with execution operation of EU, which improves the speed of execution of the instructions.

Syllabus Topic : Memory Segmentation

1.5 Memory Segmentation

→ **(MSBTE - S-15, W-15, W-16, S-17, W-17)**

Q. 1.5.1 Describe memory segmentation in 8086 microprocessor and list its four advantages. (Ref. sec.1.5) S-15, 4 Marks

Q. 1.5.2 What is memory segmentation? How it is done in 8086 microprocessor? (Ref. sec. 1.5) W-15, 4 Marks

Q. 1.5.3 Explain the concept of segmentation in 8086. (Ref. sec 1.5) W-16, S-17, W-17, 4 Marks

- The memory in an 8086 based system is organized as segmented memory and this memory management technique is called as segmentation.
- The complete physically memory is divided into a number of logical segments in segmentation.
- Size of each segment is 64 Kbytes and addressed by one of the segment register i.e. CS, DS, ES or SS.
- The 16-bit content of the segment register holds the starting address of a particular segment.
- So, we need an offset address or displacement or effective address to address a specific memory location within a segment.
- *The offset address is 16-bit so the maximum offset value will be FFFF H and hence the maximum size of any segment is 2^{16} = 64k locations.*
- The CPU 8086 is able to address 1 Mbytes of physical memory. The complete 1 Mbytes memory can be divided into 16 segments, each of 64 kbytes size as shown in Fig. 1.5.1.
- The address of the segments may be assigned as 0000H to F000H respectively.
- The offset address values are from 0000H to FFFFH so that the physical addresses range from 00000H to FFFFFH.

- The segments register contains the higher order 16 bits of the starting addresses for four memory segments i.e. Data segment, Code segment, Stack segment, Extra segment that the 8086 CPU works with at a particular time.

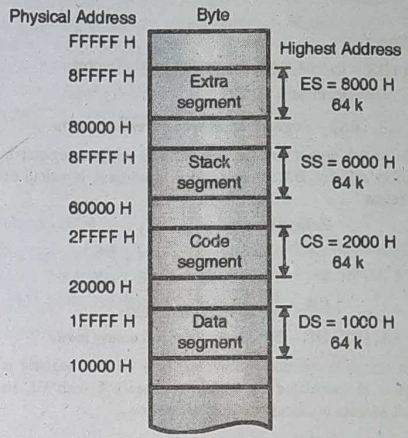


Fig. 1.5.1 : Memory segment

- As shown in Fig. 1.5.1, the base address is nothing but the starting addresses of each segment, for example, the starting base address of data segment is 10000H, 20000H for code segment etc.
- The 16-bits offset or displacement is added to the 16-bits segment base register after shifting the contain of it toward left by one digit to get 20-bits physical address.

Advantages of segmentation

1. The address associated with any instruction or data is only 16-bits though the 8086 has 20-bits physical address.
2. Segmentation can be used in multi-user time shared system.
3. Programs and data can be stored separately from each other in segmentation.
4. We can have program of more than 64 kbytes or data more than 64 kbytes by using more than one code or data segments.
5. Segmentation makes it possible to write programs which are position independent or dynamically re-locatable.
6. Segmentation allows two processes to share data.
7. Segmentation allows you to extend the addressability of a processor.

Syllabus Topic : Physical Memory Address Generation

1.6 Physical Memory Address Generation

→ **(MSBTE - W-14, S-15, W-16, S-17, W-17)**

Q. 1.6.1 Describe the physical address generation process I 8086 microprocessor. (Ref. sec. 1.6) W-14, 1 Mark

Q. 1.6.2 Describe how 20 bit physical address is generated in 8086 microprocessor. Give one example. (Ref. sec. 1.6) S-15, 4 Marks

Q. 1.6.3 Describe how 20 bit physical address is formed in 8086 micro processors with one suitable example. (Ref. sec. 1.6) W-16, 4 Marks

Q. 1.6.4 Give the steps in physical address generation in 8086 microprocessor. (Ref. sec. 1.6) S-17, 2 Marks

Q. 1.6.5 With the help of diagram, describe physical memory address generation of 8086. (Ref. sec. 1.6) W-17, 4 Marks

- One Mbyte (1024 Kbyte) of physical memory can be interface with the 8086 microprocessor because 8086 has 20 address lines i.e. 2^{20} = 1024 Kbyte or 1 Mbyte
- The segment registers are used to hold 16-bit of the starting address of four memory segments.
- But 8086 has 20-bit address bus, so it can address any of 2^{20} = 1 Mbytes in memory.
- The address associated with any instruction or data byte is only 16-bit called as effective address or offset or displacement or logical address.
- The logical addresses are used to calculate physical address.

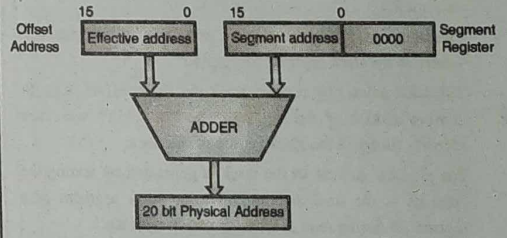


Fig. 1.6.1 : Physical address generation

- However, the address outputted by BIU is 20-bit called as physical address; Fig. 1.6.1 shows how 8086 calculates physical address from the effective address.

Other registers in this section are given as follows :

Instruction Pointer IP

- The instruction pointer register holds the 16-bit address of the next code byte within the code segment.
- The value stored in IP is called as offset or displacement.
- This offset is added to the code segment register after shifting it by four bits, which include base address of the code segment. For example, let us take CS = 3000 H and IP = 0123 H.
- Fig. 1.6.2 shows the calculation of physical address from CS and IP.

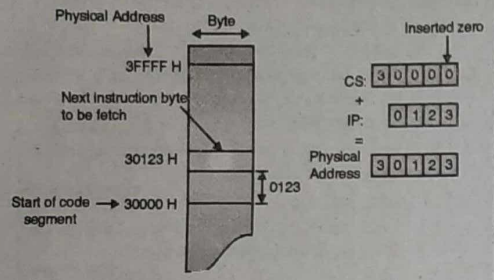


Fig. 1.6.2 : Physical address generation for Code Segment

- Before adding the content of CS and IP, the content of CS is shifted left by four bits position.
- Now CS contains 3000 H, when shifted left by four bit positions, it gives 30000 H, which is base address of the code segment i.e. start address.
- When we add offset 0123 H, the 30123 H becomes the 20-bit physical address of the next instruction byte to be fetched.
- 20-bit physical address is normally represented as CS:IP.

Stack pointer SP

- The 8086 allows us to set aside an entire 64 kbytes segment as a stack.
- The upper 16-bit of the starting address of this segment is loaded in the stack segment SS register.
- The stack pointer SP register holds the 16-bit offset from the starting address of the segment where the word was most recently stored on the stack i.e. top of the stack.
- The physical address of the stack is generated by adding the contents of the stack pointer SP to the stack segment base register SS during read or write operation with stack.
- So, the content of SS stack base segment register is shifted left by four bits and then the content of SP register is added to it. For example, if SS contains 6000 H and SP contains FFE0 H. The SS is shifted left four bit position to give 60000 H.
- After adding SP i.e. offset in to it, the resultant physical address for the top of the stack will be 6FFE0 H as shown in Fig. 1.6.3. This can be represented as SS:SP.

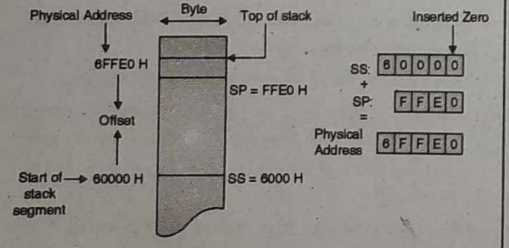


Fig. 1.6.3 : Physical address generation of stack

Example 1.6.1

If BX = 0158H, DI = 10A5H, DS = 2100H and displacement is 1B57H and DS is used as segment register then calculate physical address produced for different memory addressing mode.

Solution :

Given : BX = 0158H, DI = 10A5H and DS = 2100H

Displacement = 1B57H

MOV AX, [BX] – Register Indirect addressing mode

In above instruction the default base address register is DS and offset register BX, so the physical address is calculated as given below

Zero is inserted				↓		
	2	1	0	0	0	DS
+		0	1	5	8	BX
P.A.=	2	1	1	5	8	

MOV AX, [BX+DI] – Base Indexed Addressing mode

In above instruction the default base address register is DS and offset is calculated by adding register BX with DI, so the physical address is calculated as given below

Zero is inserted				↓		
	2	1	0	0	0	DS
+		0	1	5	8	BX
+		1	0	A	5	DI
P.A.=	2	2	1	E	D	

MOV AX, [1B57] – Direct addressing mode

In above instruction the default base address register is DS and offset is given directly in instruction, so the physical address is calculated as given below

Zero is inserted				↓		
	2	1	0	0	0	DS
+		1	B	5	7	16 bit offset
P.A.=	2	2	B	5	7	

Example 1.6.2

If DS = 345AH and SI = 13DCH, calculate physical address.

Solution :

Suppose instruction using SI as a offset register is MOV AX, [SI] in Register Indirect addressing mode

In above instruction the default base address register is DS and offset register SI, so the physical address is calculated as given below

Zero is inserted				↓		
	3	4	5	A	0	DS
+		1	3	D	C	SI
P.A.=	3	5	9	7	C	

Example 1.6.3

What is displacement? How does it determine memory address in a MOV [2000H], CL instruction?

Solution :

The instruction pointer register holds the 16-bit address of the next code byte within the code segment. The value stored in IP is called as offset or displacement. Assume DS= A000H, Memory address in MOV [2000H], CL is given below

Zero is inserted				↓		
	A	0	0	0	0	DS
+		2	0	0	0	Displacement
P.A.=	A	2	0	0	0	

Example 1.6.4

State the default segment base and offset pair register. If CS = 1000H and IP = 2000H, then from which memory location 8086 reads an instruction.

Solution :

The default segment base and offset pair register are CS:IP and SS:SP.

Zero is inserted				↓		
	1	0	0	0	0	CS
+		2	0	0	0	IP
P.A.=	1	2	0	0	0	

After calculating physical address, then 8086 reads an instruction from memory address 12000H in a code segment.

Example 1.6.5

Calculate the physical address generated by

(i) 4370 : 561E (ii) 7A32 : 0028

Solution :

(i)

Zero is inserted				↓		
	4	3	7	0	0	Base
+		5	6	1	E	Offset
P.A.=	4	8	D	1	E	

(ii)

Zero is inserted				↓		
	7	A	3	2	0	Base
+		0	0	2	8	Offset
P.A.=	7	A	3	4	8	

Example 1.6.6

IF DS = C239H and SI = 8ABCH, then Calculate physical address.

Solution :

Zero is inserted				↓		
	C	2	3	9	0	CS
+		8	A	B	C	IP
P.A.=	1	A	4	4	C	

Example 1.6.7 S-14, 4 Marks

Calculate the physical address in the following cases.

(a) CS (b) DS

Solution :

(a) CS : 1200H and IP DE00H the physical address is calculated below

Zero is inserted				↓		
	1	2	0	0	0	SS
+		D	E	0	0	SP
P.A.=	1	F	E	0	0	

(b) DS : 1F00H and BX : 1A00H for MOV AX, [BX]

Zero is inserted				↓		
	1	F	0	0	0	SS
+		1	A	0	0	SP
P.A.=	2	0	A	0	0	

Example 1.6.8 W-14, 1 Mark

If CS = 69FAH and IP = 834CH, calculate the physical address generated.

Solution :

Zero is inserted				↓		
	6	9	F	A	0	CS
+		8	3	4	C	IP
P.A.=	7	2	2	E	C	

Example 1.6.9 S-17, 4 Marks

Calculate the physical address for given.

(i) DS = 73A2H SI = 3216H

(ii) CS = 7370H IP = 561EH

Solution :

(i)

Zero is inserted				↓		
	7	3	A	2	0	DS
+		3	2	1	6	SI
P.A.=	7	6	C	3	6	

(ii)

Zero is inserted				↓		
	7	3	7	0	0	CS
+		5	6	1	E	IP
P.A.=	7	8	D	1	E	

Example 1.6.10 S-18, 2 Marks

Calculate the physical address for the given

CS = 3420H, IP = 689AH.

Solution :

Zero is inserted				↓		
	3	4	2	0	0	CS
+		6	8	9	A	IP
P.A.=	3	A	A	9	A	

Syllabus Topic : Concepts of the Pipelining

1.7 Concepts of the Pipelining

➔ **(MSBTE - S-14, W-14, S-15, W-15, S-16, S-17, W-17, S-18)**

Q. 1.7.1 State the concept of pipelining of 8086. (Ref. sec. 1.7) S-14. 2 Marks

Q. 1.7.2 Define pipeline. (Ref. sec. 1.7) W-14. 1 Mark

Q. 1.7.3 What is the size of instruction pre-fetch queue in 8086 microprocessor ? (Ref. sec. 1.7) W-14. 1 Mark

Q. 1.7.4 Explain the pipelining in 8086 microprocessor. How is queuing useful in speeding up the operation of 8086 microprocessor. (Ref. sec. 1.7) S-15. 4 Marks

Q. 1.7.5 What is pipelining? State its need and how it is done in 8086 ? (Ref. sec. 1.7) W-15. 4 Marks

Q. 1.7.6 What is pipelining? How it is implemented in 8086 microprocessor. (Ref. sec. 1.7) S-16. 2 Marks

Q. 1.7.7 Explain the concept of pipelining in 8086 microprocessor with diagram. (Ref. sec. 1.7) S-17. 4 Marks

Q. 1.7.8 Describe concept of pipelining in 8086. (Ref. sec. 1.7) W-17. 4 Marks

Q. 1.7.9 State the advantages of pipeline architecture. (Ref. sec. 1.7) S-18. 4 Marks

- The technique used to enable an instruction to complete with each clock cycle is called as pipelining.
- Normally, on a non-pipelined processor, nine clock cycles are required for fetch, decode and execute cycles for the three instructions as shown in Fig. 1.7.1(a).
- But, on a pipelined processor, the fetch, decode and execute operation are performed **in parallel**, only five clock cycles are required to execute the same three instructions as shown Fig. 1.7.1(b).
- First instruction requires three cycles to complete the execution.
- Next instructions then complete at a rate of one instruction per cycle.
- During the clock cycle 5 we have I_3 completing, I_4 being decoded and I_5 being fetched as shown in Fig. 1.7.1(b).
- Stack is a reserved area of the memory in the RAM, where temporary information may be stored.
- Stack operates on the principle of Last In First Out (LIFO).
- The queue operates on the principle of first in first out (FIFO).
- So that the execution unit gets the instruction for execution in the order they fetched.
- Feature of fetching the next instruction while the current instruction is executing is called pipelining which will reduce the execution time.
- So, pipelining improve the execution speed of the processor.
- In 8086, pipelining is implemented by providing 6 byte queue where as long as 6 one byte instructions can be stored well in a advance and then one by one instruction goes for decoding and execution.
- So, while executing first instruction in a queue, processor decodes second instruction and fetches 8th instruction from the memory.
- In this way, 8086 perform fetch, decode and execute operation in parallel i.e. in single clock cycle as shown in Fig. 1.7.1(b).

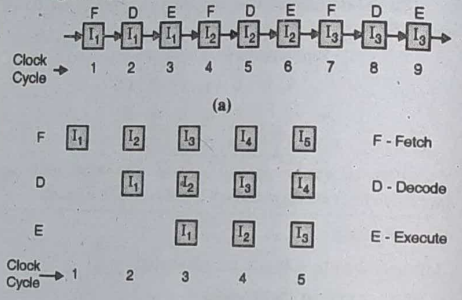


Fig. 1.7.1 : Pipelined execution of three instructions

Advantages of Pipelining

- Pipelining enables many instructions to be executed at the same time.
- It allows execution to be done in fewer cycles.
- Speed up the execution speed of the processor.
- More efficient use of processor.

1.8 Differences between Minimum and Maximum Mode Operation of 8086

➔ **(MSBTE - S-14, W-15, W-16, S-18)**

Q. 1.8.1 Compare maximum and minimum mode configuration of 8086. (any four points) (Ref. sec. 1.8) S-14, W-16. 4 Marks

Q. 1.8.2 Differentiate between minimum mode and maximum mode of 8086 microprocessor (Eight points). (Ref. sec. 1.8) W-15, S-18. 4 Marks

Sr. No.	Minimum mode	Maximum mode
1.	MN / $\overline{MX}$ pin is connected to Vcc.	MN / $\overline{MX}$ pin is grounded.
2.	No separate bus controller is required.	Separate bus controller (8288) is required.
3.	Control signals M/ $\overline{IO}$, $\overline{RD}$, $\overline{WR}$ are available on 8086 directly.	Control signals M / $\overline{IO}$, $\overline{RD}$, $\overline{WR}$ are not available on 8086 directly but status of the control signals are available on status pins $\overline{S_0}$, $\overline{S_1}$ and $\overline{S_2}$.
4.	Control signals such as $\overline{IOR}$, $\overline{LOW}$, $\overline{MEMW}$, $\overline{MEMR}$ can be generated using control signals M/$\overline{IO}$, $\overline{RD}$, $\overline{WR}$ are available on 8086 directly.	Control signals such as, $\overline{MRDC}$, $\overline{MWTC}$, $\overline{AMWC}$, $\overline{IORC}$, $\overline{LOWC}$, $\overline{AIOWC}$, are generated by bus controller 8288 using status signals $\overline{S_0}$, $\overline{S_1}$ and $\overline{S_2}$.
5.	ALE, $\overline{DEN}$, DT/ $\overline{R}$ and $\overline{INTA}$ signals are directly available.	ALE, DEN, DT/R and $\overline{INTA}$ signals are not directly available and are generated by bus controller 8288.
6.	HOLD and HLDA signals are available to interface another master in system such as DMA controller.	$\overline{RQ}$ / GT_0 and $\overline{RQ}$ / GT_1 signals are available to interface another master in system such as DMA controller and Co-processor 8087.
7.	Status of the instruction queue is not available.	Status of the instruction queue is available on pins QS_0 and QS_1.

Chapter Ends...

❑❑❑

CHAPTER 2

UNIT - II

The Art of Assembly Language Programming

Syllabus

Program development steps : Defining problem and Constrains, Writing Algorithms, Flowchart, Initialization checklist, Choosing instructions, Converting algorithms to assembly language programs.

Assembly Language Programming Tools : Editors, Assembler, Linker, Debugger, Assembler directives.

2.1 Introduction

- As compare to the high-level programming, assembly language programming is slightly cumbersome.
- The programs written in assembly language are compact and efficient.
- A program has to be converted to machine code for execution, so it is performed by the translator called as Assembler.
- Assembly language programming requires good knowledge of machine architecture, operating system and programming principles.
- Assembly language is case insensitive, so program can be loaded either in uppercase, lowercase or combination of lowercase and uppercase.
- The program development tools such as editor, assembler, linker, and debugger are required for the programming.

Syllabus Topic : Program Development Steps

2.2 Program Development Steps

➔ **(MSBTE - W-15, S-17)**

Q. 2.2.1 State the steps involved in program development. (Ref. sec. 2.2) W-15. 4 Marks

Q. 2.2.2 List the program development steps for assembly language programming. (Ref. sec. 2.2) S-17. 2 Marks

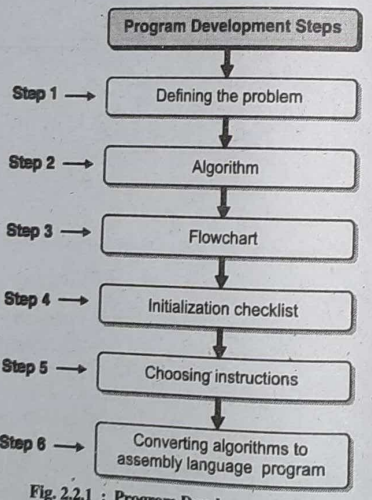


Fig. 2.2.1 : Program Development Steps

Syllabus Topic : Defining Problem

2.2.1 Defining the Problem

- The first step while writing program is to define very carefully about the problem that you want the program to solve.
- At this point you need not to write down program but you must know what you would like to do.

Syllabus Topic : Algorithm

2.2.2 Algorithm

➔ **(MSBTE - S-14, W-14, S-16, W-16, W-17)**

Q. 2.2.3 What is algorithm ? (Ref. sec. 2.2.2) S-14. W-14. S-16. W-16. W-17. 1 Mark

- The formula or sequence of operations or tasks need to perform by your program can be specified as a step in general English and is often called as Algorithm.
- In short, an *algorithm* is a step by step method or statement written in general English language of solving a problem

Syllabus Topic : Flowchart

2.2.3 Flowchart

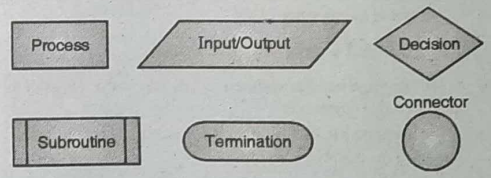
➔ **(MSBTE - S-14, W-14, S-16, W-16, W-17, S-18)**

Q. 2.2.4 What is flowchart? Sketch any four symbols use in flowchart. State their use. (Ref. sec. 2.2.3) S-14. 3 Marks

Q. 2.2.5 Define flowchart. (Ref. sec. 2.2.3) W-14. S-16. W-16. W-17. 1 Mark

Q. 2.2.6 Draw the symbols used in a flowchart while developing ALP. Mention the use of each symbol. (any 4).(Ref. sec. 2.2.3) S-18. 4 Marks

- The flowchart is a graphically representation of the program operation or task.
- The specific operation or task is represented by graphical symbol such as circle, rectangle, diagonal, square and parallelogram etc. given below :



Syllabus Topic : Initialization Checklist

2.2.4 Initialization Checklist

- In program there are many variables, constants and various part of the system such as segment registers, flags, stack, programmable ports etc. which must be initialize properly.
- The best way to approach the initialization task is to make the checklist of the entire variables, constants, all the registers, flags and programmable ports in the program.
- At this point you will come to know which parts on the checklist will have to be initialized.

Syllabus Topic : Choosing Instructions

2.2.5 Choosing Instructions

- Next step is to choose appropriate instruction that performs your problem's operations or tasks.
- This is an important step, so you must know entire instruction set of the microprocessor, the operation performed and flag affected after the execution by the instructions.

Syllabus Topic : Converting Algorithms to Assembly Language Programs

2.2.6 Converting Algorithms to Assembly Language Program

- Once you have selected the instructions for the operations to be performed, then arrange these instructions in sequence as per algorithm, so that desired output must be obtaining after execution.
- In assembly language program, a first step is to set up and declare the data structure that the algorithm will be working with, then write down the instruction required for initialization at the start of the code section.
- Next determine the instructions required to implement the major actions in the algorithm and declare how data must be positioned for these instructions.

Syllabus Topic : Assembly Language Program Development Tools

2.3 Assembly Language Program Development Tools

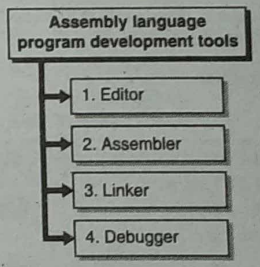


Fig. 2.3.1 : Assembly language program development tools

2.3.1 Editors

➔ **(MSBTE - W-14, W-15)**

Q. 2.3.1 State the function of Editor. (Ref. sec. 2.3.1) W-14. W-15. 2 Marks

- An editor is a program which helps you to construct your assembly language program in right format so that the assembler will translate it correctly to machine language.
- So, you can type your program using editor.
- This form of your program is called as source program.
- The **DOS** based editor such as **EDIT**, **WordStar**, and **Norton Editor** etc. can be used to type your program.

2.3.2 Assembler

➔ **(MSBTE - S-14, W-14, S-15, W-15, S-16, S-17, S-18)**

Q. 2.3.2 State the function of Assembler. (Ref. sec. 2.3.2) S-14, W-14, S-15, W-15, S-16, S-17, S-18, 2 Marks

- An assembler is a program that translate assembly language program to the correct binary code fro each instruction i.e. machine code and generate the file called as object file with extension **.obj.**
- Some examples of assembler are **TASM** Borland's Turbo Assembler and **MASM** Microsoft Macro Assembler etc.

2.3.3 Linker

➔ **(MSBTE - S-15, S-16, W-16, W-17)**

Q. 2.3.3 State the function of Linker.(Ref. sec. 2.3.3) S-15, S-16, W-17, 2 Marks

Q. 2.3.4 Describe Linker with respect to their function and usages. (Ref. sec. 2.3.3) W-16, 2 Marks

- A linker is a program, which combines, if requested, more than one separately assembled module into one executable program, such as two or more programs and also generate .exe module and initializes it with special instructions to facilitate its subsequent loading the execution.
- Some examples of linker are **TLINK** Borland's Turbo Linker and **LINK** Microsoft's Linker etc.

2.3.4 Debugger

➔ **(MSBTE - W-16, S-17, W-17)**

Q. 2.3.5 Describe Debugger with respect to their function and usages. (Ref. sec. 2.3.4) W-16, 2 Marks

Q. 2.3.6 State the function of debugger. (Ref. sec. 2.3.4) S-17, W-17, 2 Marks

- Debugger is a program that allows the execution of program in single step mode under the control of the user.
- The process of locating and correcting errors using a debugger is known as debugging.
- Some examples of debugger are DOS **Debug** command, Borland's turbo Debugger **TD**, Microsoft Debugger known as Code View **CV** etc.

2.4 Program Development Process (PDP)

- The PDP is process, which consists of analysis, design, development, implementation, translation, testing, debugging and maintenance of the program.
- It is an interactive and iterative process that involves the following steps.

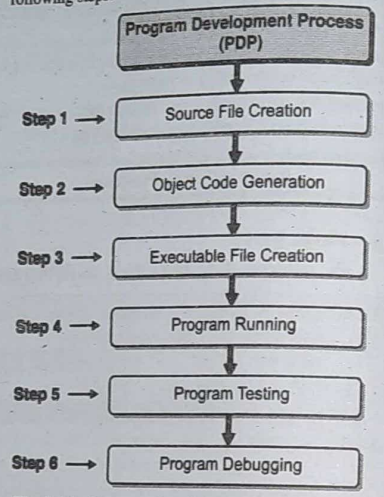


Fig. 2.4.1 : Program Development Process (PDP)

2.4.1 Source File Creation

- The file containing program statement in assembly language is known as a source file.
- The source file is created and edited using text editor and must have an extension .ASM.

2.4.2 Object Code Generation

- The language translator is used to translate source program to re-locatable object file.
- The assembler is used to translate assembly language source code to re-locatable object code.

2.4.3 Executable File Creation

Linker is used to create an executable file.

2.4.4 Program Running

- The executable file can be run by entering the name of executable file on the prompt and by pressing ENTER key on the keyboard.

2.4.5 Program Testing

- The result or output generated by the program has to be tested for their validity. The result must satisfy the program specification.
- If any errors occur in the result, then program should be debug.

2.4.6 Program Debugging

- The errors in the program can be located using debugger.
- The executable file of the program to be debugged must be created with the program debug options.

Syllabus Topic : Assembler Directives

2.5 Assembler Directives and Operator

➔ **(MSBTE - S-14, W-14, W-15)**

Q. 2.5.1 List any four assembler directives and explain any two of them. (Ref. sec. 2.5) S-14, 4 Marks

Q. 2.5.2 List any four assembler directives. State the functions of any two assembler directives. (Ref. sec. 2.5) W-14, 4 Marks

Q. 2.5.3 What are assembler directives? Explain any two assembler directives. (Ref. sec. 2.5) W-15, 4 Marks

☞ **Directives**

- Assembly language program supports a number of reserve word i.e. key word that enables you to control the way in which a program assembles and lists.
- These words are called as a **assembler directives,** act only during the assembly of the program and generate no machine executable code.
- So, directives are the statement that gives direction to the assembler and also called as pseudo-instructions that are no translated in to the machine code.
- Directives are divided into various categories.

2.5.1 Data Definition and Storage Allocation Directives

Q. 2.5.4 Describe following assembler directives.
(i) DB (Ref. sec. 2.5.1(a))
(ii) EQU (Ref. sec. 2.5.1(h)) S-15, S-18, 2 Marks

Q. 2.5.5 Explain the following assembler directives.
(1) ORG (2) EQU (3) DD
(Ref. sec. 2.5.1) S-16, 3 Marks

Q. 2.5.6 Describe the functions of the following directives :
1) DD 2) DB 3) DUP
(Ref. sec. 2.5.1) W-16, 3 Marks

Q. 2.5.7 Explain following assembler directives
(i) DB (ii) DW (iii) DD (iv) DQ
(Ref. sec. 2.5.1) S-17, 4 Marks

Q. 2.5.8 Describe the function of following directives.
(i) DD (ii) DB (iii) DUP (iv) EQU
(Ref. sec 2.5.1) W-17, 4 Marks

- Data definition directives are used to define the program variables and allocate a specified amount of memory to them.
- The data definition directives are DB, DW, DD, DQ, DT, STRUCT and RECORD.

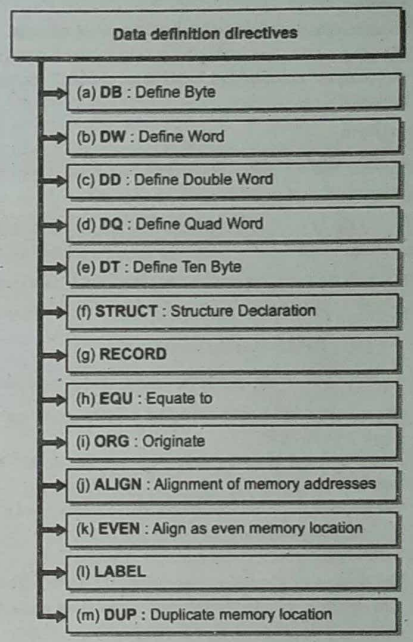


Fig. 2.5.1 : Data definition directives

➔ **(a) DB : Define Byte**

➔ **(MSBTE - S-15, W-16, S-17, W-17, S-18)**

- The directive DB is used to define a byte type variable.
- It can be used to single or multiple byte variable.
- The range of values that can be stored in a byte is 0 to 255 for unsigned numbers and –128 to +127 for signed numbers.

General form

```
Name_Of_Variable DB Initialization_Value(,s)
```

Examples

```
NUM DB ?                    ; Allocate One memory location
NAME DB' VIJAY'             ; Allocate Five memory locations
ARRAY DB 12, 25, 26, 55, 65 ; Allocate Five memory locations
LIST DB 100 DUP(0)          ; Allocate Hundred memory locations
```

(b) DW : Define Word

(MSBTE - S-17)

- The directive DW is used to define a word type i.e. 2-bytes variable.
- It can be used to define single or multiple word variables.
- The range of values that can be stored in a word is 0 to 65535 for unsigned numbers and – 32768 to + 32767 for signed numbers.

General form

Name_Of_Variable DW *Initialization_Value(,s)*

Examples

```
NUM     DW  ?            ; Allocate two memory locations
NUM     DW  '78'         ; Allocate two memory locations
TABLE DW  1, 2, 3, 564   ; Allocate ten memory locations
LIST    DW  50 DUP(0)    ; Allocate 100 memory locations.
```

(c) DD : Define Double Word

(MSBTE - S-16, W-16, S-17, W-17)

- The directive DD is used to define a double word type i.e. 4 byte type variable.
- It can be used to define single or multiple double word variables.
- The range of values that can be stored in a double word is 0 to 2^{32-1} for unsigned numbers and -2^{32-1} to $+2^{32-1}-1$ for signed integer numbers.
- The floating-point numbers range from 10^{-38} to 10^{38}. The DD type variables are used in the math coprocessor instructions where large numbers are used in computation.

General form

Name_Of_Variable DD Initialization_Value(,s)

Examples

```
NUM     DD  ?            ; Allocate four memory locations
TABLEDD     1, 2, 3, 5, 9 ; Allocate 20 memory locations
LIST    DD  10 DUP(0)    ; Allocate forty memory locations.
```

(d) DQ : Define Quad Word

(MSBTE - S-17)

- The directive DQ is used to define a quad word type i.e. 8 byte type variable.
- It can be used to define single or multiple quad word variables.
- The range of values that can be stored in a double word is 0 to 2^{64-1} for unsigned numbers and -2^{64-1} to $+2^{64-1}-1$ for signed integer numbers.
- The floating-point numbers range from 10^{-308} to 10^{308}. The DQ type variables are used in the math coprocessor instructions where large numbers are used in computation.

General form

Name_Of_Variable DQ *Initialization_Value(,s)*

Examples

```
NUM     DQ  ?            ; Allocate eight memory locations
TABLEDQ     1, 2, 3,5,9  ; Allocate forty memory Locations
LIST    DQ  10 DUP (0)   ; Allocate eighty memory
                         ; locations.
```

(e) DT : Define Ten Byte

- The directive DT is used to define a ten byte type variable.
- It can be used to define a single or multiple 10 byte variables.
- The range of values that can be stored is 0 to 2^{80-1} for unsigned numbers and -2^{80-1} to $2^{80-1}-1$ for signed integer numbers.
- The floating-point numbers range from 10^{-4932} to 10^{4932}. The DT type variables are useful in math coprocessor instructions where large numbers are used in computation.

General form

Name_Of_Variable DT *Initialization_Value(,s)*

Examples

```
NUM     DT ?             ; Allocate Ten memory
                         ; locations
TABLE DT 1, 2, 3, 5, 9   ; Allocate fifty memory
                         ; locations
LIST    DT 10 DUP(0)     ; Allocate Hundred memory
                         ; locations.
```

(f) STRUCT : Structure Declaration

- The directives STRUCT is used to declare the data type which a collection of primary data types (DB, DW, DD).
- The structure declaration allows the user to define a variable, which has more than one data type.

General form

```
Structure_Name   STRUCT
                 ...
; Sequence of DN,DW,DD directives for declaring .
; field
                 ...
Structure_Name   ENDS
```

Structure variable definition

- Structure declaration does not allocate memory space, but merely defines pattern. Storage space is allocated only when the structure variable is defined.
- The general form of a structure definition is as follows :

Variable Structure_Name<Initializations>Accessing structure variable field

- The structure variable field can be accessed by using a dot('.') operator known as the structure access operator.
- It is placed between the structure variable and the field, which has to be accessed.

- The general form of the structure variable definition is given below.
- Structure_Variable.Field_Name

Examples

The structure of EMPLOYEE is defined as follows

```
EMPLOYEE        STRUCT
EMP_NUM     DW  ?
EMP_NAME    DB  25 DUP(0)
EMP_DEPT    DB  30 DUP(0)
EMP_AGE     DB  ?
EMPLOYEE    ENDS
```

The structure variable EMPLOYEE_1, EMPLOYEE_2 etc. can be defined as follows.

EMPLOYEE_1 EMPLOYEE < 01,'MARTINE'.'COMPUTER TECH.',35>

EMPLOYEE_2 EMPLOYEE<>

The field EMP_NUM, EMP_NAME, EMP_DEPT, EMP_AGE in a structure variable EMPLOYEE_1 is initialized with the value 01, 'MARTINE', 'COMPUTER TECH.', and 35 respectively. The fields of the structure variable EMPLOYEE_2 are un-initialized. The field of the variable EMPLOYEE_1 can be accessed as follows.

EMPLOYEE_1.EMP_NUM
EMPLOYEE_1.EMP_NAME
EMPLOYEE_1.EMP_DEPT
EMPLOYEE_1.EMP_AGE

The instruction MOV AL, EMPOYEE_1.EMP_NUM will load AL with the contents of EMP_NUM field .

(g) RECORD

- The directive RECORD is used to define a bit pattern within a byte or a word.
- It is similar to the bit-wise access in C language.
- The RECORD definition helps in encoding or decoding of bit for which some meaning is assigned.

General form

```
Record_Name  RECORD
Field_Specification_1.....Field_Specification_N
```

Where each field specification is of the form, Field_Name : Length [= Initialization] where initialization is optional.

Examples

In serial communication, initializations of the port parameter are coded bit-wise as follows.

```
Bit 7, 6, 5     : Baud Rate
Bit 4,3         : Parity
Bit 2           : Stop Bits
Bit 1,0         : Word Length
```

Using RECORD directive, the above byte can be coded as follows.

SERIAL_COM **RECORD** BAUD:3, PARITY : 2, STOP:1, WORDLENGTH:2

The instruction MOV AL, BAUD will place the bits 7, 6, 5 of variable SERIAL_COM in the register AL as bits 2, 1, 0. The instruction MOV BAUD, AL will place the bits 2, 1, 0 of the variable SERIAL_COM in the register AL as bits 7, 6, 5.

(h) EQU : Equate to

(MSBTE - S-15, S-16, W-17, S-18)

- The EQU directive is used to declare the symbols to which some constant value is assigned.
- Such symbols are called as macro symbols, so macro assembler will replace every occurrences of the symbol in a program by its value.
- Macros are also used to increase the readability of a program.
- The advantage of macros that the modification of the symbol value at the declaration will be reflected throughout the program.

General form

Symbol Name EQU *Expression*

Examples

```
NUM         EQU     100
INCREAMENT  EQU     INC
START_STR   EQU     [SI]
```

(i) ORG : Originate

(MSBTE - S-16)

- The directives ORG assigns the location counter with the value specified in the directive.
- It helps in placing the machine code in the specified location while translating the instructions into machine codes by the assembler.
- This feature is useful in building device drivers and .COM programs whose structure is explicit as certain information has to be located in a definite place in the program.

General form

ORG [$+] Numeric_value

Examples

```
ORG     100H
ORG     $
ORG     $+100
```

(j) ALIGN : Alignment of memory addresses

The directive ALIGN is used to force the assembler to align the next data item or instruction according to given value.

General form

ALIGN Numeric_Value

The number must be a power of 2 such as 2, 4, 8 or 16.

Examples

ALIGN 4

- In above statement, the assembler advances its location counter to the next address that is evenly divisible by 4.
- If the location counter is already at the required address, then it is not advanced.
- The assembler fills unused bytes with zeros for data and NOP for instructions.

➜ (k) EVEN : Align as even memory location

Q. 2.5.9 Describe assembler directive : EVEN. (Ref. sec. 2.5.1(k))

- The directive EVEN is used to inform the assembler to increment the location counter to the next even memory address.
- If it is location counter is already pointing to even memory address, it should not be incremented.
- The 80x86 processor reads a word from the memory in one bus cycles while accessing an even memory address word.
- It requires two bus cycles to access a word from the odd memory location.
- The even alignment with EVEN directives helps in accessing a series of consecutive memory word quickly.
- The directive can be used in both code and date segments which increment the location counter to the next even memory location if necessary.
- The use of EVEN in the code segment is actually replaced by the instruction by NOP.

General form

```
EVEN
```

Examples

```
DATA SEGMENT
Name    db    'VIJAY$'
EVEN
AGE     db    30
DATA    ENDS
```

➜ (l) LABEL

- The directive LABEL enables you to redefine the attribute of a data variable or instrucfion label.

General form

```
Variable_Name   LABEL   Type_Specifier
```

Examples

```
TEMP        LABEL  BYTE
NUM LABEL   WORD
```

➜ (m) DUP : Duplicate memory location

➜ (MSBTE - W-16, W-17)

- The DUP directive can be used to generate multiple bytes or words with known as well as un-initialized values.

Example

```
table dw    100   DUP(0)
stars db    50    dup('*')
ARRAY3 DB   30    DUP(?)
```

2.5.2 Program Organization Directives

Q. 2.5.10 Describe following assembler directives.
(i) ASSUME (Ref. sec. 2.5.2 (a))
(ii) SEGMENT (Ref. sec. 2.5.2 (b))
S-15, S-18, 2 Marks

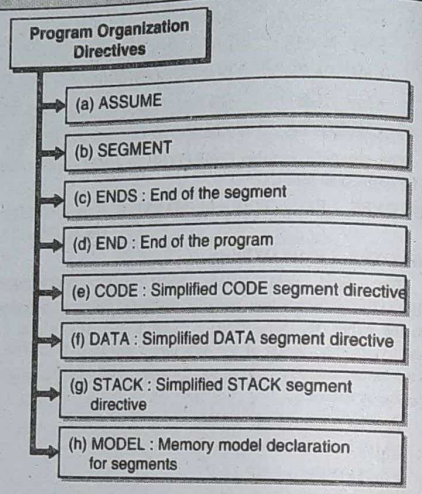


Fig. 2.5.2 : Program Organization Directives

- The 8086 programs are organized as a collection of logical segment.
- The directives used to organize the program segments are SEGMENT, END, ASSUME etc.
- The segment can enclose program data, code or both.

➜ (a) ASSUME

➜ (MSBTE - S-15, S-18)

- The directive ASSUME informs the assembler the name of the logical segment that should be used for a specified segment.
- When program is loaded, the processor segment register should point to the respective logical segments.

General form

```
ASSUME
Seg_Reg: Seg_Name,....., Seg_Reg:Seg_Name
```

Where,

- ASSUME is a assembler directive.
- Seg_Reg is any of the segments register i.e. CS, DS, ES, SS.
- Seg_Name is the name of an user defined segment and must be any valid symbol except reserved keywords.

Examples

```
ASSUME CS:My_Code, DS:My_Data, S:My_Stack,
ES:My_Extra
```


In the above statement, ASSUME is a directive, CS is a code segment register and symbol My_Code is an user defined name.

➜ (b) SEGMENT

➜ (MSBTE - S-15, S-18)

Q. 2.5.11 Describe assembler directive : SEGMENT. (Ref. sec. 2.5.2(b))

- The directive SEGMENT is used to indicate the beginning of the logical segment.
- The directive SEGMENT follows the name of the segment.
- The directive SEGMENT and ENDS must be enclosed the segment data, code, extra or stack of program.

General form

```
Segment_Name   SEGMENT   [WORD/PUBLIC]
```

- The use of the type specifier WORD indicates that the segment has to be located at the next available address, otherwise, the segment will be located at the next available paragraph (16-byte size) which might waste up to 15 bytes of the memory.
- The type specifier PUBLIC indicates that the given segments, which have the same name.

Examples

```
(a)  My_Data SEGMENT
     .....
     Program Data Definition Here
     .....
     My_Data ENDS
(b)  My_Code SEGMENT
     .....
     Program Code Here
     .....
     My_Code ENDS
```

➜ (c) ENDS : End of the segment

Q. 2.5.12 Describe the meaning of the directives : ENDS. (Ref. sec. 2.5.2(c))

- The directive ENDS informs the assembler the end of the segment.
- The directive ENDS and SEGMENT must enclosed the segment data or code of the program.

General form

```
Segment_Name   ENDS
```

Examples

```
(a)  My_Data SEGMENT
     .....
     Program Data Definition Here
     .....
     My_Data ENDS
(b)  My_Code SEGMENT
     .....
     Program Code Here
     .....
     My_Code ENDS
```

➜ (d) END : End of the program

Q. 2.5.13 Describe the assembler directive : END. (Ref. sec. 2.5.2(d))

The directive END is used to inform assembler the end of the program.

General form

```
END [Start_Address]
```

- The optional start_address specifies the location in the code segment where execution is to be start.
- The system loader uses this address to load CS register.

➜ (e) .CODE : Simplified CODE segment directive

- This simplified segment directive defines the code segment.
- All executable code must be placed in this segment.

General form

```
.CODE [name]
```

➜ (f) .DATA : Simplified DATA segment directive

- This simplified segment directive defines the data segment for initialized near data.
- All data definition and declaration must be placed in this segment.

General form

```
.DATA
```

➜ (g) .STACK : Simplified STACK segment directive

This simplified segment directive define the stack segment and default size of the stack is 1024 bytes, which you may override.

General form

```
.STACK 100
```

➜ (h) .MODEL : Memory model declaration for segments

This simplified segment directive creates default segments.

General form

```
.MODEL memory_small
```

☞ Memory models

- **TINY :** Since MASM 6.0, used for .COM program.
- **SMALL :** All data in one segment and all code in one segment.

- **MEDIUM :** All data in one segment, but code in more than one segment.
- **COMPACT :** Data in more than one segment, but code in one segment.
- **LARGE :** Both data and code in more than one segment, but no array may exceed 64 kbytes.
- **HUGE :** Both data and code in more than one segment, and array may exceed 64 kbytes.

2.5.3 Value Returning Attribute Directives

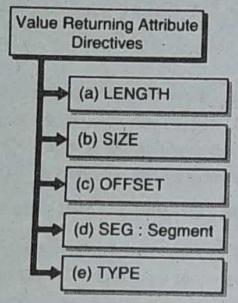


Fig. 2.5.3 : Value Returning Attribute Directives

- The task of programming can be made easier by assigning the assembler to computer the size of the data items.
- It is performed using the directives LENGTH, SIZE, OFFSET and TYPE and causes the assembler to substitute some numeric constant depending on the data item.

➜ (a) LENGTH

- The directive LENGTH informs the assembler about the number of the elements in a data items such as array.
- If the array is defined with DB, then it returns number of bytes allocated to a variable.
- If an array is defined with DW, then it returns the number of word allocated to the array variable.

General form

LENGTH *Variable_Name*

Examples

```
MOV CX, LENGTH NAME
MOV DX, LENGTH ARRAY
```

➜ (b) SIZE

The directives SIZE is same as LENGTH except that it returns the number of bytes allocated to the data item instead of the number of elements in it.

General form

SIZE *Varaible_Name*

Examples

```
MOV AX, SIZE NAME
MOV DX, SIZE TOTAL
```

➜ (c) OFFSET

- The directive OFFSET informs the assembler to determine the displacement of the specified variable with respect to the base of the segment.
- It usually used to load a offset of a variable into the register.
- Using this offset value, a variable can be referenced using indexed addressing modes.

General form

OFFSET *Variable_Name*

Examples

```
MOV SI, OFFSET ARRAY
MOV DX, OFFSET MSG
MOV BX, OFFSET NUM
```

➜ (d) SEG : Segment

The directive SEG is used to determine the segment in which the specified data items is defined.

General form

SEG *Variable_Name*

Examples

```
MOV DS, SEG MSG
MOV ES, SEG LIST
```

➜ (e) TYPE

- The directive TYPE is used to determine the type of the data item.
- It determines the number of byte allocated to the data TYPE.
- Assembler allocates one byte for DB, two byte for DW and four byte for DD type variable definition.

General form

TYPE *Variable_Name*

Examples

```
ADD BX, TYPE NUM
SUB DX, TYPE PI
```

2.5.4 Procedure Definition Directives

The procedure definition directives are used to define subroutines. It offers modular programming constructs like PROC and ENDP.

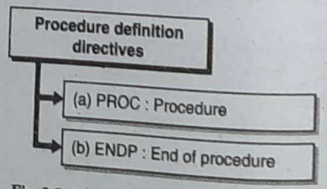


Fig. 2.5.4 : Procedure definition directives

➜ (a) PROC : Procedure

- The directive PROC indicates beginning of a procedure and follows with the name of the procedure.

- The term FAR or NEAR follows the PROC directive indicating the type of a procedure.
- If the term is not specified, then assembler assumes NEAR as the type specifier.
- The use of a procedure type specifier helps the assembler to decide whether to code RET as near return or far return.
- The directive PROC is used with the directive ENDP to enclose the procedure code.

General form

Procedure_Name PROC *[NEAR/FAR]*

Example

```
ADD     PROC    NEAR
        .....
        .....
        Procedure Codes
        .....
        .....
ADD     ENDP
```

- The above procedure can be called by using CALL instruction of 80x86 microprocessor whenever required such as CALL ADD.
- From above example, it is clear that the procedure will save a great amount of effort and time by avoiding the overhead of writing the repeated pattern of code.

➜ (b) ENDP : End of procedure

- The directive ENDP informs the assembler the end of a procedure.
- The directive ENDP and PROC must enclose the procedure code.

General form

Procedure_Name ENDP

Examples

```
FACTORIAL   ENDP
HEXTOASC    ENDP
```

2.5.5 Macro Definition Directives

The macro definition directives are used to define macro functions. The directives macro and ENDM are used in the definition of the macro.

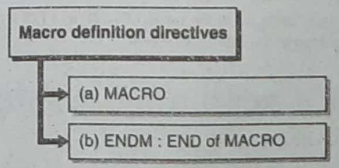


Fig. 2.5.5 : Macro definition directives

➜ (a) MACRO

- The directive MACRO informs the assembler the beginning of a macro.
- It consists of name of a macro followed by keyword MACRO and macro arguments if any.
- The directives MACRO and ENDM must enclose the definition, declaration or a small part of code, which have to be substituted at the invocation of a macro.

General form

Macro_Name MACRO *[Argument1,……..,ArgumentN]*

Example

```
DISP MACRO MSG
     PUSH AX
     PUSH DX
     MOV AH,09H
     LEA DX,MSG
     INT 21H
     POP DX
     POP AX
     ENDM
```

- The above macro whose name is DISP can be called by writing macro name along with its argument if any in the program wherever it is required i.e, number of times.
- The macro function are called as an *open subroutine*, macro gets expanded if a call is made to it.
- The difference between macros and procedures is that, a call to macro will be replaced with its body during assembly time whereas the call to the procedure will be an explicit transfer of program control to the called procedure during run time.
- The concept of macro is illustrated below.

```
.DATA
MSG1 DB   'Well Com to Computer Department$'
MSG2 DB   'Hardware Laboratory $'
.CODE
.....
DISP MSG1
DISP MSG2
.....
ENDS
END
```

For above example, it is clear that the macros will save a great amount of effort and time by avoiding the overhead of writing the repeated pattern of code.

➜ (b) ENDM : END of MACRO

- The directive ENDM informs the assembler the end of the macro.
- The directive MACRO and ENDM must enclose the definition, declarations, or a small part of the code which have to be substituted at the invocation of the macro.

General form

ENDM

Example

```
DISP MACRO MSG
     PUSH AX
     PUSH DX
     MOV AH,09H
     LEA DX, MSG
     INT 21H
     POP DX
     POP AX
     ENDM
```

2.5.6 Data Control Directives

The data control directives are used to declare the variable used in communication of information between the program modules. The data control directives are PUBLIC, EXTERN and PTR.

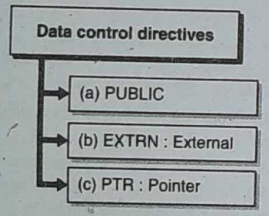


Fig. 2.5.6 : Data control directives

➜ (a) PUBLIC

- The directive PUBLIC informs the assembler that the specified variable or segment can be accessed from other program modules.
- It helps in managing the multiple program modules by sharing the global variables or procedures.
- The variable and procedure to be shared must be declared as PUBLIC in a module in which it physically exists.

General form

PUBLIC *variable1, variable2,.....variableN*

Example

```
PUBLIC MSG, NAME, NUM
PUBLIC ARRAY
```

➜ (b) EXTRN : External

The directive EXTRN informs the assembler that the data items or label following the directive will be used in a program module, which is defined in the other program modules.

General form

For variable reference

EXTRN

varaibel_name1:refernce_type,.....variable_nameN:reference_type

For procedure reference

EXTRN *procedure_name:[NEAR/FAR]*

Examples

```
EXTRN msg:byte, name:word, num:byte
EXTRN DISPLAY:NEAR.
EXTRN DISPLAY:FAR
```

➜ (c) PTR : Pointer

- The directive PTR is used to indicate the type of the memory access i.e. BYTE / WORD / DWORD.
- For instance, if the assembler encounters the instruction like INC [SI], it will not be able to decide whether to code for byte increment or word increment.
- It can be solver using PTR directive with the instruction as INC BYTE PTR [SI] for byte increment or INC WORD PTR [SI] for word increment.
- The directive PTR can be used to override the type declaration of a variable i.e. if a variable is declared as word, it can be accessed as a byte.

Examples

```
INC BYTE PTR [DI].
ADD AL, BYTE PTR NUM.
DEC WORD PTR [BX].
```

2.5.7 Branch Displacement Directives

The branch displacement directives are used to control branch displacement and these are SHORT and LABEL.

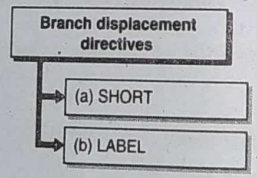


Fig. 2.5.7 : Branch displacement directives

➜ (a) SHORT

- The directive SHORT informs the assembler that the one byte displacement is required to code to jump instruction.
- Normally, two bytes are reserved to store the target address in the jump instruction.
- The target must be in the range of –128 to +127 bytes from the address of the Jump instruction.

Examples

```
JMP SHORT NEXT.
```

➜ (b) LABEL

- The directive LABEL assigns a name to the current value in the location counter.
- The LABEL directive must be followed by the definition of data type to which label is associated.
- If the label is used as a destination in the jump or call instruction, the label must be specified as a far or near.

- When the label is used to reference the data item, the label must be specified as type BYTE, WORD or DWORD.

General form

LABEL *Label_name Label_Type*

Examples

```
LABEL A_REF WORD
A DW 1000
The label A_REF can be used as reference to the variable A.
NEXT_NUM        LABEL FAR
NEXT_NUM:
                  ::
                  ::
LOOP NEXT_NUM
```

2.5.8 File Inclusion Directive

➜ **(MSBTE - W-16)**

Q. 2.5.14 Explain the assembler directive : INCLUDE. (Ref. sec. 2.5.8) **W-16, 1 Mark**

The file inclusion directive is used to define include file header and directive in INCLUDE.

INCLUDE

- The directive INCLUDE informs the assembler to include the statement defined in the include file.
- The name of the include file follows the statement INCLUDE.
- It is used to place all the data and frequently used macros into a file known as header or include file.
- This include file must exist in the directory specified in the path specification otherwise, and then assembler gives an error.
- The part of assembler which processes the include file is known as pre-processor.

General form

INCLUDE <file path specification with file name>

Example

```
INCLUDE C:\TASM\MACRO.LIB
INCLUDE C:\TASM\BIN\MYPROC.LIB
```

2.5.9 Target Machine Code Generation Control Directive

- The assembler will recognized only 8086 instructions by default.
- It is because the program that confirm themselves to the 8086 instruction set that can run on any IBM PC, irrespective of the microprocessor i.e. 8086, 80186, 80286 etc.
- So, special directive can be used at the beginning of a program to inform to the assembler to generate a code for the specific processor and these are listed below.

General form

```
.186 - generate machine code for 80186 processor only.
.286 - generate machine code for 80286 processor only.
.386 - generate machine code for 80386 processor only.
.486 - generate machine code for 80486 processor only.
.586 - generate machine code for 80586 processor only
       i.e. pentium
```

2.6 Difference between Assembler Directive and Instructions

Sr. No.	Assembler directive	Instructions
1.	Assemble directives gives direction to assembler	Instruction performs operation specified by it using processor.
2.	Assembler directives does not exeeute	Instruction are executed by the processor.

Chapter Ends...

❑❑❑

CHAPTER 3

UNIT - III

Instruction Set of 8086 Microprocessor

Syllabus

Machine Language Instruction format, Addressing modes. Instruction set, Groups of Instructions : Arithmetic Instructions; Logical Instruction, Data transfer instructions, Bit manipulation Instructions, String Operation Instructions, Program control transfer or branching Instructions, Process control Instructions.

3.1 Introduction

- In chapter 2, we have studied the art of assembly language programming for 8086 microprocessors.
- This chapter introduces the different format of instructions, addressing mode, groups of instructions and instruction set of 8086 microprocessors.
- 8086 has more than 20,000 instructions.

Syllabus Topic : Machine Language Instruction Format

3.2 Instruction Format

➔ **(MSBTE - S-14)**

Q. 3.2.1 Write the instruction format of 8086 microprocessor. (Ref. sec. 3.2)

Q. 3.2.2 List instruction formats of 8086 and explain any one. (Ref. sec. 3.2) **S-14, 4 Marks**

- A machine language instruction format has one or more numbers of fields associated with it.
- The first field is called as operation code field or op-code field, which indicate the type of the operation to be performed by the CPU.
- The second field is called as operand field i.e. data field on which CPU perform the operation specified by the instruction op-code.
- 8086 instruction set has six general formats of the instructions.
- The length of an instruction may vary from one byte to six bytes.
- The instruction formats are discussed below :

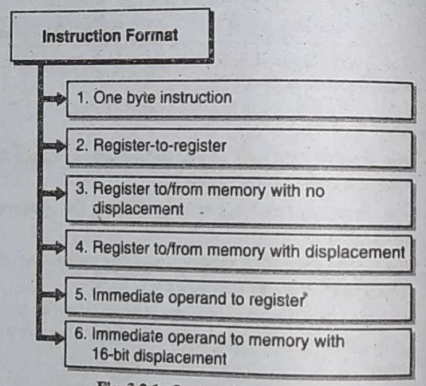


Fig. 3.2.1 : Instruction Format

➔ 1. One-byte instruction

- This format is only one byte long and may have implicit data or register operands.
- The least significant 3 bits of the op-code are used to specify the register operand, if any.
- Otherwise, all the 8 bits form an op-code and the operands are implied.

➔ 2. Register-to-register

- Register to register instruction, the format of instruction is of 2 bytes long.
- The first byte of code indicates operation code of the instruction i.e. op-code and width of the operand specified by W bit.
- The second byte of the instruction code indicates the register operand and R/M field as shown below.
- The register specified in the REG field is one of the register operands.

- The R/M field indicate another operand which may be register or memory location.

1st byte

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
Op-Code							W

2nd byte

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
1	1	REG			R/M		

➔ 3. Register to/from memory with no displacement

- In this type of instruction, the format of instruction is of 2 bytes long.
- The first byte is same as in the register-to-register format but second byte contains MOD field as shown in below.
- The MODE, R/M, REG and the W field are given from Table 3.2.1.

1st byte

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
Op-Code							W

2nd byte

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
MOD		REG			R/M		

➔ 4. Register to/from memory with displacement

- This type of instruction format includes one or two additional bytes for displacement along with two bytes' format of register to/from memory as shown as follows.

1st byte

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
Op-Code							W

2nd byte

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
MOD		REG			R/M		

3rd byte

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
Lower byte displacement							

4th byte

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
Higher byte displacement							

➔ 5. Immediate operand to register

- In this type, the instruction format content the first bytes as well as the 3-bits from the second byte, which are used for REG field, if it is a register-to-register format, are used for op-code.
- In this type, instruction format includes one or two bytes of immediate data as shown below.

1st byte

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
OPCODE							

2nd byte

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
1	1	OPCODE			R/M		

3rd byte

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
Lower byte DATA							

4th byte

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
Higher byte DATA							

➔ 6. Immediate operand to memory with 16-bit displacement

- The length of this type of instruction format is five or six-byte long.
- The first two consecutive bytes contain the information of OP-CODE, MODE and R/M fields.
- The last four bytes contain two bytes of displacement and two bytes of data shown as follows.

1st byte

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
OPCODE							

2nd byte

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
MOD		OPCODE			R/M		

3rd byte

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
Lower byte of displacement							

4th byte

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
Higher byte of displacement							

5th byte

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
Lower byte DATA							

6th byte

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
Higher byte DATA							

The op-code of instruction is usually appearing in the first byte, but in some instructions, a register destination is in the first byte and some other instructions may have their 3-bits of op-code in the second byte. The op-code have single bit indicators and their definitions and significances are given below.

- **W-bit :** Indicates the width of operands i.e. 8-bits or 16-bits data. If W = 0, the operand is of 8-bits and if W = 1, the operand is of 16-bits.
- **D-bit :** Indicates one of the operand is register in case of two operand instructions. If D bit =0, the register specified by the REG field is source operand else it is a destination operand.
- **S-bit :** It is sign extension bit and is used with W-bit to show the type of operation i.e. either byte or word operation.
- **V-bit :** Used in case of shift and rotates instructions. If shift count is 1, this bit is set to 0 and if CL contains the shift count more than 1, this bit is set to 1.
- **Z-bit :** Used by REP instruction to control the looping operation.

The REG code of the different registers in the op-code bytes are assign with binary codes given in Table 3.2.1.

Table 3.2.1

W-bit	Register address bits	Registers (8/16 bit)	Segment register (2 bit code)	Segment register (16 bit)
0	000	AL	00	ES
0	001	CL	01	CS
0	010	DL	10	SS
0	011	BL	11	DS
0	100	AH		
0	101	CH		
0	110	DH		
0	111	BH		
1	000	AX		
1	001	CX		
1	010	DX		
1	011	BX		
1	100	SP		
1	101	BP		
1	110	SI		
1	111	DI		

- First the addressing mode of the instruction must be decided, to find out the MOD and R/M fields of a particular instruction. The addressing mode depends on the operands and states how the effective address may be calculated for locating the operand, if it store in memory. The different addressing modes are listed in Table 3.2.2.
- The R/M and addressing mode row content indicates the R/M field and addressing mode specifies the MOD field.

Table 3.2.2

Operands	Memory operands			8 /16 bit Register operands	
	without offset	With 8-bits Offset	With 16-bits Offset	11	
MOD / R/M	00	01	10		
				W=0	W=1
000	(BX) + (SI)	(BX) + (SI) + D8	(BX) + (SI) + D16	AL	AX
001	(BX) + (DI)	(BX) + (DI) + D8	(BX) + (DI) + D16	CL	CX
010	(BP) + (SI)	(BP) + (SI) + D8	(BP) + (SI) + D16	DL	DX
011	(BP) + (DI)	(BP) + (DI) + D8	(BP) + (DI) + D16	BL	BX
100	(SI)	(SI) + D8	(SI) + D16	AH	SP
101	(DI)	(DI) + D8	(DI) + D16	CH	BP
110	D16 (Direct address)	(BP) + D8	(BP) + D16	DH	SI
111	(BX)	(BX) + D8	(BX) + D16	BH	DI

Note : D8 and D16 represent 8 and 16 bit displacements respectively.

- When a data is referred as an operand, then DS is the default data segment register, CS is the default code segment for storing program codes, SS is the default stack segment register and ES is the default segment register for the destination data storage.

Syllabus Topic : Addressing Modes of 8086

3.3 Addressing Modes of 8086

➔ **(MSBTE - S-14, W-14, S-15, W-15, S-16, W-16, S-17, W-17)**

Q. 3.3.1 State example of immediate addressing mode. (Ref. sec. 3.3) **S-14, S-16, 2 Marks**

Q. 3.3.2 Define addressing mode. List any two addressing mode of 8086 microprocessor. (Ref. sec. 3.3) **W-14, 2 Marks**

Q. 3.3.3 Describe various addressing modes of 8086 with one suitable example each. (Ref. sec. 3.3) **S-15, 4 Marks**

Q. 3.3.4 State and explain any four addressing modes of 8086 microprocessor with example. (Ref. sec. 3.3) **W-15, 4 Marks**

Q. 3.3.5 Explain following addressing modes of 8086 with example Immediate addressing mode (Ref. sec. 3.3) **W-16, 2 Marks**

Q. 3.3.6 Define immediate and direct addressing mode. Also give one example of each. (Ref. sec. 3.3) **S-17, 2 Marks**

Q. 3.3.7 List any two addressing modes of 8086 with example. (Ref. sec. 3.3) **W-17, 2 Marks**

- Addressing modes is used to locate data or operands in memory, register or I/O.
- Any instruction of 8086 can be used in one or more addressing modes but some instruction cannot be used in any of the addressing modes depending on the data type used in the instruction and the memory-addressing mode.
- Hence, the addressing modes of any instruction specifies the type of operands and the way they are accessed for executing an instruction. According to the flow of instruction execution, the instructions can be categorized as : Sequential control flow instructions and Control transfer instructions.
- Sequential control flow instructions are the instructions which transfer the control to the next instruction appearing immediately after it in the program after execution, e.g. the arithmetic, logical, data transfer and process control instructions.
- The control transfer instructions transfer the control to some predefined address of the memory which may or may not be specified in the instruction, after their execution. For example, INT, CALL, JMP, RET etc.

☞ **Addressing modes of 8086**

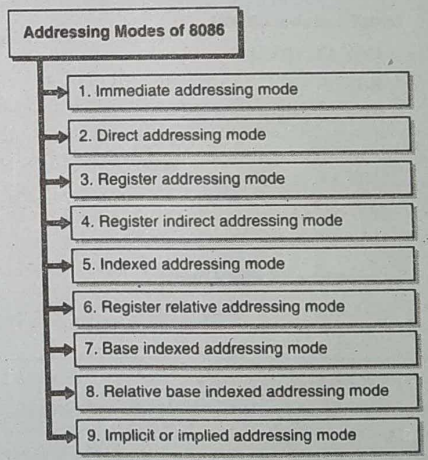


Fig. 3.3.1 : Addressing Modes of 8086

➔ 1. Immediate addressing mode

- In this mode, the immediate data is the part of the instruction and appear in the form of successive byte or bytes after the op-code bytes.
- So immediate data may be 8 bit [byte] or 16 bit [word] in length.
- Immediate data can be accessed quickly as they are available in an instruction queue hence no extra bus cycle is required to read data.

Examples

MOV AL, 46H	AL is loaded with 8-bit immediate data 46H.
MOV BX, 1234H	BX is loaded with 16-bit immediate data 1234H.

➔ 2. Direct addressing mode

- In this mode, a 16-bits memory address (offset) of operand is directly specified in the instruction as a part of it.
- The offset of displacement may be either 8 bit or 16 bit which follows the instruction op-code.
- So, the physical address is calculated by adding this offset to the base segment registers i.e. CS, DS, ES, SS.

Examples

MOV AL, [3000H]	AL will be loaded with the content of memory location whose offset is 3000H from base address.
AND AX, [8000H]	AX will be ANDed with the content of memory location whose offset is 8000H from the base.

- In above examples, DS is the default base address register.
- Suppose, data is stored in EXTRA segment, then data can be loaded in register by specifying address using following way.

MOV AX, ES:[4000H]

➔ 3. Register addressing mode

- In this mode, the data is stored in a registers and it is referred using the particular register i.e. all register except IP may be used in this mode.
- Register may be source operands, destination operand or both.
- The instruction of this addressing mode are compact and faster in execution as all registers are reside in chip and no external bus is required to read data.
- Registers may be 8 bit or 16 bit.

Examples

MOV AX, CX	Copies the content of CX reg. to AX reg.
AND AL, BL	ANDing the content of BL with AL, store result in AL.
ROR AL, CL	Rotate the contents of AL CL times.

➔ 4. Register indirect addressing mode

- In this mode, the address of the memory location which contains data or operand is available in an indirect way, using offset register such as BX, SI, DI register.
- The default segment register is either DS or ES depending the instruction used.
- If BP is used, then SS is the default segment register.

Examples

MOV AX, [BX]	Copies the contents of memory location whose offset is in BX Register.

SUB [SI], AL	Subtract the content of AL from the memory location whoseoffset is in SI register and store result in same memory location.

5. Indexed addressing modes

- In this mode, the offset of the operand is stored in any one of the index registers i.e. SI and DI.
- DS and ES are the default segments for index register SI and DI respectively depending on the instruction used.
- This mode allows the use of a signed displacement.

Examples

MOV BL, [SI]	Copies the byte from memory location whose offset is in index register SI to AL.
ADD AX, [DI+8]	Copies the word from memory location whose offset will be calculated by adding 8 with the content of DI register.

6. Register relative addressing mode

- In this mode, the data is available at an effective address formed by adding 8-bits or 16-bits displacement with content of any one of the registers such as BX, BP, SI and DI in the default DS and ES segment.

Examples

MOV AX, 50[BX]	Copies the word from memory location whose offset will be calculated by adding the 50 with the content of BX register.
ADD AX, 5000[BP]	Copies the word from memory location whose offset will be calculated by adding 5000H to the content of BP register.

7. Base indexed addressing mode

- In this mode, the effective address of data is calculated by adding the content of a base register BX or BP to the content of an index register SI or DI with default segment DS or ES.

Examples

MOV AX, [BX][SI]	Copies the word from memory location whose offset is calculated by adding the content of BX with the contents of SI.
ADD AL, [BX][DI]	Copies the byte from memory location whose offset is calculated by adding the content of BX with the contents of DI.

8. Relative base indexed addressing mode

- In this mode, the offset address of data is calculated by adding the 8-bits or 16-bits displacement with the sum of base register BX or BP and SI or DI register in the default segment DS and ES.

Example

MOV AX, 60[BX][SI]	Copies word from memory location whose offset is calculated by adding the 60H with the contents of BX and SI i.e. [60+BX+SI]

9. Implicit or Implied addressing mode

- Instructions using this mode have no operands. In this instruction itself will specify the data to be operated by the instruction.

Examples

CLC	This clears carry flag to zero.
DAA	Perform operation on AL register

Example 3.3.1 S-14, 4 Marks

Write appropriate instructions to perform following operations :

(a) Initialize stack at 42000H

(b) Rotate register BX right 4 times

Solution :

(a) Initialize stack at 42000H

```
MOV AX, 4000H
MOV SS, AX
MOV SP, 2000H
   OR
MOV AX, 4200H
MOV SS, AX
MOV SP, 0000H
```

(b) Rotate register BX right 4 times

```
MOV CL, 4
ROR BX, CL
```

Example 3.3.2 W-14, 4 Marks

What will be the contents of AX register after execution of following lines of code.

```
MOV AL, 10
MOV DL, 20
MUL DL
```

Solution :

- **MOV AL, 10 :** This instructions loads 0AH in AL register as 10 is considered as decimal number as H is written at the end of number.
- **MOV DL, 20 :** This instructions loads 14H in DL register as 10 is considered as decimal number as H is written at the end of number.

- **MUL DL :** This instruction store result 00C8H in AX register which is nothing but 200 in decimal.

Example 3.3.3 W-14, 4 Marks

Identify the addressing mode of the following instructions.

(i) MUL AL, BL (ii) MOV AX, BX

(iii) MOV BX, [SI] (iv) MOV DX, 0040H

Solution :

```
(i)   MUL AL, BL          ; Register addressing mode
(ii)  MOV AX, BX          ; Register addressing mode
(iii) MOV BX, [SI]        ; Indexed addressing mode
(iv)  MOV DX, 0040H       ; Immediate addressing mode
```

Example 3.3.4 W-14, 4 Marks

What will be the contents of AL, BL, AX, DX register after the execution of all 4 lines.

```
MOV AL, 03H
MOV BL, 03H
SUB AL, BL
MUL AL, 08H
```

Also write which flags will be affected?

Solution :

(1) MOV AL, 03H instruction loads 03H in AL register.

(2) MOV BL, 03F instruction loads 03H in BL register.

(3) SUB AL, BL instruction subtract BL from AL, as numbers in AL and BL are same so result will be 0 in AL and ZF will set.

(4) MUL AL, 08H instruction should be written as MUL 08H and this instruction does not execute because immediate addressing mode for this instruction is not allowed i.e. 08H cannot be used.

Example 3.3.5 W-14, 4 Marks

Write assembly language instructions to perform following operations.

(i) Move contains of memory location pointed by DI into BX register

(ii) Copy a string from one memory location to another using string manipulation instruction.

Solution :

(i) Move contains of memory location pointed by DI into BX register

```
MOV BX, [DI]
```

(ii) Copy a string from one memory location to another using string manipulation instruction.

```
MOV CX, count      ; Load length of string as a count
MOV SI,offset str_s ; Initialize memory pointer for
                    ; source string
MOV DI,offset str_d ; Initialize memory pointer for
                    ; destination; string
REP  MOVSB
```

OR

```
MOV CX, count      ; Load length of string as a count
MOV SI,offset str_s ; Initialize memory pointer for
                    source string
MOV DI,offset str_d ; Initialize memory pointer for
                    destination string
UP:  MOVSB
LOOP UP
```

Example 3.3.6 S-15, 4 Marks

Identify addressing modes in following instructions.

(i) MOV AX, 2050H (ii) STC

(iii) MOV AL, DS:[SI] (iv) INC BX

Solution :

```
(i)   MOV AX, 2050H       ; Immediate addressing mode
(ii)  STC                 ; Implicit addressing mode
(iii) MOV AL, DS:[SI]     ; Indirect/indexed addressing mode
(iv)  INC BX              ; Register addressing mode
```

Example 3.3.7 S-15, 4 Marks

What will be content of register BX after the execution of instructions.

```
MOV BX,  050H
MOV CL,  05H
SHL  BX, CL
```

Solution :

	Shift left → BX Register = 2050H																
Out	D_{15}	D_{14}	D_{13}	D_{12}	D_{11}	D_{10}	D_9	D_8	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	CL=5
	0	0	1	0	0	0	0	0	0	1	0	1	0	0	0	0	
0	0	1	0	0	0	0	0	0	1	0	1	0	0	0	0	0	5
0	1	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	4
1	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	3
0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	2
0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	1
	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	Result
	Finally result in BX = 0A00H after shifting five times toward left																

Example 3.3.8 S-14, 4 Marks

Identify the addressing mode used in following Instructions:

(a) MOV DS, AX (b) MOV AX, [4172H]

(c) ADD AX, [SI] (d) ADD AX, [SI][BX][04]

Solution :

(a) MOV DS, AX : Register addressing mode
(b) MOV AX, [4172H] : Direct addressing mode
(c) ADD AX, [SI] : Indirect or indexed addressing mode
(d) ADD AX,[SI][BX][04] : Relative base index addressing mode

Example 3.3.9 W-15, 4 Marks

Identify the addressing modes in following instructions

(a) MUL AL, BL (b) MOV AX, 2100H

(c) MOV AL, DS : [SI] (d) MOV AX, BX

Solution :

(a) MUL AL, BL : Register Addressing Mode
(b) MOV AX, 2100H : Immediate Addressing mode
(c) MOVAL, DS:[SI] : Indexed or Indirect Addressing mode
(d) MOV AX, BX : Register Addressing Mode

Example 3.3.10 S-16, 4 Marks

Identify the addressing mode of following instructions.

(i) INC [4712H] (ii) ADD AX, 4712H

(iii) DIV BL (iv) MOV AX. [BX + SI]

Solution :

(i) INC [4712H] : Direct addressing mode
(ii) ADD AX, 4712H : Immediate addressing mode
(iii) DIV BL : Register Addressing mode
(iv) MOV AX [BX + SI] : Base Indexed Addressing mode

Example 3.3.11 S-17, 4 Marks

Identify the addressing mode of following instructions.

(i) MOV AX, 2034H
(ii) MOV AL, [6000H]
(iii) ADD AL, CL
(iv) MOV AX, 50H [BX] [SI]

Solution :

(i) MOV AX, 2034H : Immediate addressing mode
(ii) MOV AL, [6000H] : Direct addressing mode
(iii) ADD AL, CL : Register addressing mode
(iv) MOV AX, 50H [BX] [SI] : Relative base index addressing mode

Example 3.3.12 S-18, 4 Marks

Identify the addressing modes for the following instruction :

(i) MOV CL, 34 H
(ii) MOV BX, [4172 H]
(iii) MOV DS, AX
(iv) MOV AX, [SI + BX + 04]

Solution :

(i) MOV CL, 34 H ; Immediate addressing mode
(ii) MOV BX, [4172 H] : Direct addressing mode
(iii) MOV DS, AX : register addressing mode
(iv) MOV AX, [SI + BX + 04] : Base Index with 8 bit displacement

Example 3.3.13 S-16, 4 Marks

Analyze the content of AL register and status of carry and auxiliary carry flag after the execution of following instructions.

MOV AL, 99H ADD AL, 01H DAA

Solution :

If AL = 99 BCD and add AL with 01 BCD

Then ADD AL, 01H

```
  1001 1001  =  AL = 99 BCD
+ 0000 0001  =  BL = 01 BCD
  1001 1010  =  AL = 9A H and
       CF    =  0, AF=0
```

Now, in above example after addition, Carry and Auxiliary carry flags are reset but lower and higher nibble is greater than or equal to 9. So DAA instruction adds 6 to higher as well as lower nibble of AL register to get correct BCD result i.e. 100 BCD of which 00 in AL and Cy = 1 as given below.

After the execution of DAA instruction, the result is

```
Cy = 1   1001 1010  =  AL = 9A H AF = 1
       + 0110 0110  =  66H
         0000 0000  =  AL = 00 in BCD form
                    =  100 in BCD        ...Ans.
```

Syllabus Topic : Instruction Set

3.4 Instruction Set of 8086

Q. 3.4.1 Select instructions for each of the following :
i. Rotate register BL, right 4 times
ii. Multiply AL by 08H
iii. Signed division of BL and AL
iv. Move 5000H to register DS.
(Ref. sec. 3.4)

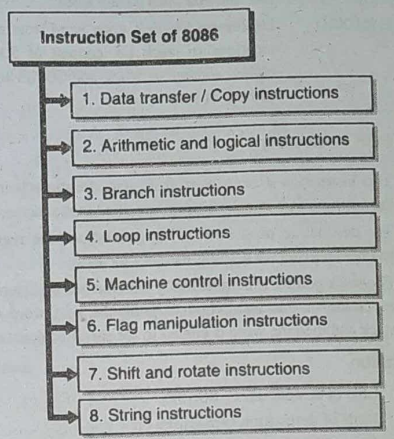


Fig. 3.4.1 : Instruction Set of 8086

The 8086 instructions are grouped into following main types.

➜ 1. Data transfer instructions group

The instructions of this group are used to transfer data from source to destination where source may be register, memory location or immediate data and destination may be register or memory location. All the instruction which performs the store, move, load, exchange, input and output instructions comes in this category.

➜ 2. Arithmetic and logical instructions group

The instructions of this group are used to perform arithmetic and logical, increment, decrement, compare and scan operation.

➜ 3. Branch instructions group

The instructions of this group are used to transfer the program execution control to the address specified in the instruction such as call, jump and return instructions.

➜ 4. Loop instructions group

If these instructions use REP instruction prefix with CX used as count register, they can be used to perform unconditional and conditional loops. The LOOP, a LOOPNZ and LOOPZ instruction belongs to this category.

➜ 5. Process (Machine) control instructions group

The instructions of this group controls the status of machine. NOP, HLT, WAIT and LOCK instructions are the example of this type.

➜ 6. Bit (Flag) manipulation instructions group

The instructions of this group, which directly affect the flag register, such as CLD, STD, CLI, STI etc.

➜ 7. Shift and rotate instructions group

The instructions of this group is used to perform bit-wise shifting or rotation in either direction with or without a count in CX.

➜ 8. String instructions group

The instructions of this group are used to perform various string manipulation operations such as load, move, scan, compare, store etc.

Syllabus Topic : Data Transfer Instruction

3.4.1 Data Copy / Transfer Instructions

➜ (MSBTE - S-15, W-15, S-16, W-16, S-17, W-17, S-18)

Q. 3.4.2 Differentiate between instructions : MOV and LXI (Ref. sec. 3.4.1) S-15, S-18, 1 Mark

Q. 3.4.3 Explain the following instruction of 8086 : XLAT. (Ref. sec. 3.4.1) W-15, S-16, 2 Marks

Q. 3.4.4 Explain the functions of following instruction with one example : (1) XLAT (2) LEA (Ref. sec. 3.4.1) W-16, 2 Marks

Q. 3.4.5 With suitable example explain following instruction XCHG.(Ref. sec. 3.4.1) W-16, 1 Mark

Q. 3.4.6 Explain the following instruction of 8086 : (i) XLAT (ii) XCHG (Ref. sec. 3.4.1) S-17, S-18, 4 Marks

Q. 3.4.7 With example, describe XLAT instruction. (Ref. sec. 3.4.1) W-17, 4 Marks

☞ MOV destination, source

- This instruction is used transfers data from source i.e. register/memory location / immediate data to destination i.e. another register/memory location.
- The source of an instruction can be any one of the segment register or other general or special purpose register or a memory location and destination of an instruction can be a register or memory location.
- But, in immediate addressing mode, segment register should not be a destination register means direct loading of the segment registers with immediate data is not allowed.
- To load segment registers with immediate data, we must have to load any general-purpose register with the immediate data and then it should be moved to that any segment register.

Operation

Destination ← Source

Examples

MOV BX, 3456H	Immediate addressing mode where 3456H is immediate data copied to BX register by this instruction.
MOV AL, [3000H]	Direct addressing mode where the data from memory location 3000H is copied to AL register by this instruction.
MOV AX, BX	Register addressing mode where the contents of BX register is copied to AX register by this instruction.
MOV AL, [SI]	Indirect addressing mode where the data from memory location addressed by SI register by this instruction.
MOV AH, 50H[BX]	Base register relative addressing mode where the displacement 50 H is added to BX register to get effective address from which data is copied to AH register by this instruction.
MOV AL, [BX][SI]	Base index addressing mode where the effective address is calculated by adding the contents of BX and SI register and data is copied to AL register.
MOV AL, 50H[BX][DI]	Relative base indexed addressing mode where the effective address is computed by adding the displacement 50H to the sum of the content of BX and DI registers, and data is copied to AL register.

☞ PUSH source

- This instruction is used to store word from source on to the stack locations.
- Stack pointer is decremented and then stores a word from source to the location in the stack segment where the stack pointer points.
- The source of the word must be a 16-bit general-purpose register, a segment register, or 16-bit memory location.
- After decrementing the stack addresses the higher byte copies to the higher address and the lower byte copies to the lower address.

Operation

SP ← SP –2

SS:[SP] ← MSB of source

SS:[SP – 1] ← LSB of source

Examples

PUSH BX	Decrement SP by 2, copy BX to stack i.e. content of BH register to higher address of stack and BL register to lower address of stack.
PUSH DS	Decrement SP by 2, copy DS to stack i.e. higher byte of DS register to higher address of stack and lower byte of DS register to lower address of stack.
PUSH AL	Not allowed must push a word.
PUSH [5000H]	Decrement SP by 2, copy word from memory locations to stack i.e. content of 5000H to higher address of stack and 5001H to lower address of stack.

☞ POP destination

- This instruction stores a word from stack locations pointed by the stack pointer to a destination specified in the instruction.
- The destination must be a 16-bit general-purpose register, a segment register, or a 16-bit memory location.
- The stack pointer is automatically incremented by 2 during the execution of this instruction to point the next word on the stack and then the word is copied to the specified destination.

Operation

LSB of destination ← SS:[SP].

MSB of destination ← SS:[SP+1].

SP ← SP + 2.

Examples

POP DX	Copy a word from top of stack to DX, SP = SP + 2 i.e. content of [SP] to DL register and content of [SP+1] to DH register.
POP DS	Copy a word from top of the stack to DS register, SP = SP + 2.
POP [8000]	Copy a word from top of stack to memory locations 8000H and 8001H.

☞ XCHG destination, source

- This instruction exchanges the contents of a register with the contents of another register or memory location.
- The instruction cannot directly exchange the contents of two memory locations.
- A memory location can be specified as the source or as the destination by any of 24 addressing modes given in Table 3.2.2.
- The source and destination must both be word or they must both be byte. The segment register cannot be used in this instruction.

Operation performed

Destination ↔ Source

Examples

XCHG AX, BX	Exchange the word in AX with word in BX.
XCHG BL, CH	Exchange the byte in BL with byte in CH.
XCHG AX, [7000H]	Exchange the word in AX with memory i.e. AH with the content of 7000H memory location and AL with the content of 7001H memory location.

XCHG AL, NUM[BX]	Exchange the AL with byte in memory at EA = NUM[BX].

☞ IN accumulator, port

- The IN instruction copies data from a port to destination which may be AL or AX i.e. accumulator.
- The address of the port can be specified in the instruction directly or indirectly.
- For the fixed port type, the 8-bit address of a port is specified directly in the instruction.
- For variable port type the 16-bit address of a port is specified in DX register only.
- So DX register must always be loaded with the 16-bits port address before the IN instruction.

Operation

AL ← [port] for byte.

AL ← [port] and AH ← [port+1] for word

Examples

In fixed port type

IN AL, 80H	Input a byte from port whose address is 80H.
IN AX, 80H	Input a word from port whose address is 80H.

In variable port type

MOV DX, 8000H	Initialize DX to point port with port address.
IN AL, DX	Input a byte from 8-bit port whose address is in DX to AL.
IN AX, DX	Input a word from 16-bit port whose address is in DX to AX.

☞ OUT port, accumulator

- The OUT instruction copies a byte from AL or a word from AX to the specified port.
- The address of the port can be specified in the instruction directly or indirectly.
- For the fixed port type, the 8-bit address of a port is specified directly in the instruction.
- For variable port type the 16-bit address of a port is specified in DX register only.
- So DX register must always be loaded with the 16-bits port address before the OUT instruction.

Operation

[port] ← AL for byte.

[port] ← AL and [port+1] ← AH for word.

Examples

In fixed port type

OUT 80H, AL Copy the contents of AL to port 80H.

OUT 80H, AX Copy the contents of AX to port 80H.

In variable port type

MOV DX, 6000H Initialize DX with 16-bit port address.

OUT DX, AL Copy the contents of AL to port.

OUT DX, AX Copy the contents of AX to port.

☞ XLAT

- The XLAT instruction replaces a byte in the AL register with a byte from a lookup table in memory.
- Before the execution of the XLAT instruction the lookup table containing the values for the new code must be put in memory and the offset of the starting address of the lookup table must be loaded in BX.
- To point to desired byte in the lookup table the XLAT instruction adds the byte in AL to the offset of the start of the table in BX.
- It then copies the byte from the address pointed to by [BX+AL] back into AL. XLAT changes no flags.

Operation

AL ← DS:[BX+AL]

Example

```
.DATA
      TABLE    DB    '0123456789ABCDEF'
      CODE DB   11
. CODE
        .
        .
        .
      MOV BX, offset TABLE     Point BX to the start of
                               lookup table in DS
      MOV AL, CODE
      XLAT                     Replace code in AL with
                               code from lookup table. The
                               content of AL will be 0BH
```

☞ LEA 16-bit register, source

- This instruction determines the offset of the variable or memory location names as the source and loads this offset in the specified 16-bits register.

Operation

16 bit register ← effective address

Examples

LEA BX, ARRAY	Load BX with the offset of variable ARRAY.
LEA SI, LIST	Load SI with the offset of variable LIST.
LEA CX, [BX][DI]	Load CX with effective address by adding contents of BX and DI.

☞ LDS / LES 16-bit register, memory address of first word

- These instructions copy a word from two consecutive memory locations into the register specified in the instruction.
- It then copies a word from next two consecutive memory location into the DS register.

Operation

For LDS instruction

16 bit register ← [memory address]

DS ← [memory address + 2]

For LES instruction

16 bit register ← [memory address]

ES ← [memory address + 2]

Examples

LDS BX, [1234H]	Copy the contents of memory location 1234H in BL, contents of 1235H to BH and the contents of 1236H and 1237H in DS register.
LES BX, [1234H]	Copy the contents of memory location 1234H in BL, contents of 1235H to BH and the contents of 1236H and 1237H in ES register.

☞ LAHF

- This instruction stores lower byte of flag register of 8086 to the AH register.

Example : LAHF

☞ SAHF

- This instruction copies the content of AH register which is used to set or reset the flag in the lower byte of the flag register of 8086.

Example : SAHF

☞ PUSHF

- This instruction is used to store flag register on to stack.
- The stack pointer is decremented by two and stores the word in the flag register to the memory locations pointed by the stack register.

Example : PUSHF

☞ POPF

This instruction is used to store a word from the memory locations at the top of stack to the flag register and increment stack pointer by two.

Example : POPF

Syllabus Topic : Arithmetic Instructions

3.4.2 Arithmetic Instructions

➔ **(MSBTE - S-14, W-14, S-15, W-15, S-16, W-16, W-17, S-18)**

Q. 3.4.8 Compare the following instructions (2 points) : AAA and DAA. (Ref. sec. 3.4.2) **S-14, 2 Marks**

Q. 3.4.9 Write any two arithmetic instructions with their functions. Give the syntax with one example each. (Ref. sec. 3.4.2) **W-14, 2 Marks**

Q. 3.4.10 Explain the instruction of 8086 microprocessor with their syntax : ADD. (Ref. sec. 3.4.2) **W-14, 2 Marks**

Q. 3.4.11 Differentiate between instructions : ADD and ADC. (Ref. sec. 3.4.2) **S-15, S-18, 1 Mark**

Q. 3.4.12 Explain the following instruction of 8086 : DAA (Ref. sec. 3.4.2) **W-15, 2 Marks**

Q. 3.4.13 Explain DAA instruction with suitable example. (Ref. sec. 3.4.2) **S-16, W-17, 4 Marks**

Q. 3.4.14 Explain the following instruction of 8086 with suitable example : AAA. (Ref. sec. 3.4.2) **S-16, W-17, 4 Marks**

Q. 3.4.15 With suitable example explain following instructions. (i) DAA (ii) ADC (iii) MUL (Ref. sec. 3.4.2) **W-16, 3 Marks**

Q. 3.4.16 Explain following instructions : INC (Ref. sect. 3.4.2) **W-17, S-18, 2 Marks**

Q. 3.4.17 Explain with suitable example the instruction given below : (i) DAA (ii) AAM (Ref. sec. 3.4.2) **S-18, 4 Marks**

These instructions perform the arithmetic operations, like addition, subtraction, multiplication and division along with the respective ASCII and decimal adjust instructions.

☞ ADD / ADC destination, source

- The ADD instruction adds a number from some source to a number from some destination.
- The ADC instruction adds the carry flag into the result.
- The source may be an immediate number, a register, or a memory location as specified by any 24 addressing modes given in Table 3.2.2.
- The destination may be a register or a memory location specified by any one of 24 addressing modes in Table 3.2.2.
- The source and destination must be of the same type and cannot both be memory locations.
- Destination should not be an immediate number.

Flag affected : OF, CF, PF, AF, SF, ZF.

Operation :

Destination ← destination + source for ADD.

Destination ← destination + source + CF for ADC.

Examples :

ADD AL, 74H	Immediate addressing mode instruction that adds the immediate number 74H to AL and stores result in AL.
ADD AX, BX	Register addressing mode instruction that adds the contents of BX with AX and stores result in AX.
ADD AL,[6000H]	Direct addressing mode instruction that adds the contents of memory location 6000H with AL, stores result in AL.
ADD AL, [SI]	Register indirect addressing mode instruction that adds the contents of memory location pointed by SI index register with AL and store result in AL.
ADC AX, 1234H	Immediate addressing mode instruction that adds the immediate number 1234H to AX with carry and stores result in AL.
ADC AX, BX	Register addressing mode instruction that adds the contents of BX to AX with carry and stores result in AX.

☞ SUB / SBB destination, source

- The SUB instruction is used to subtract the data in source from the data in destination and the stores result in destination.
- The SBB instruction is used to subtract the source operand and the barrow [CF], which may reflect from the result of the previous calculations, from the destination operand, and the result, is stored in destination operand.
- Source must be a register or memory location or immediate data and the destination must be a register or a memory location.
- The destination operands should not be an immediate data and the source and destination both should not be memory operands.

Flag affected : OF, CF, PF, AF, SF, and ZF.

Operation

Destination ← destination - source for SUB.

Destination ← destination - source – CF for SBB.

Examples

SUB AL, 74H	Immediate addressing mode instruction that subtracts the immediate number 74H from AL and stores result in AL.
SUB AX, BX	Register addressing mode instruction that subtracts the contents of BX from AX and stores result in AX.
SUB AL, [6000H]	Direct addressing mode instruction that subtracts the contents of memory location 6000H from AL, stores result in AL.
SUB AL, [SI]	Register indirect addressing mode instruction that subtracts the contents of memory location pointed by SI index register from AL and store result in AL.
SBB AX, 1234H	Immediate addressing mode instruction that subtracts the immediate number 1234H and barrow from AX and stores result in AX.
SBB AX, BX	Register addressing mode instruction that subtracts the contents of BX and barrow from AX and stores result in AX.

☞ INC destination

- This instruction adds 1 to the indicated destination.
- The destination can be a register or memory location specified by any one of 24 ways given in Table 3.2.2.
- Immediate data cannot be an operand of this instruction.

Flag affected : OF, PF, AF, SF, ZF.

Operation

Destination ← destination + 1

Examples

INC AX	Increment the content of AX by 1.
INC [2000H]	Increment the content of memory location 2000H by 1.
INC TEMP	Increment the byte or word named as TEMP by 1.

☞ DEC destination

- This instruction subtracts 1 from the indicated destination.
- The destination can be a register or memory location specified by any one of 24 ways given in Table 3.2.2.
- Immediate data cannot be an operand of this instruction.

Flag affected : OF, PF, AF, SF, ZF.

Operation

Destination ← destination – 1.

Examples

DEC AX	Decrement the content of AX by 1.
DEC [2000H]	Decrement the content of memory location 2000H by 1.
DEC TEMP	Decrement the byte or word named as TEMP by 1.

☞ CMP destination, source

- The CMP instruction make a comparison between a byte/word from the specified source and a byte/word from the specified destination.
- The source and destination can be an immediate data, a register or a memory location specified by one of 24 ways given in Table 3.2.2.
- However, the source and the destination cannot both be memory locations.
- The comparison is actually done by non-destructive subtraction of the source byte or word from the destination byte or word i.e. the source and the destination will not changed, but the flags will set to indicate the results of the comparison.

Flag affected : OF, CF, PF, AF, SF, ZF.

Operation

(a) If destination > source then CF = 0, ZF = 0, SF = 0.

(b) If destination < source then CF = 1, ZF = 0, SF = 1.

(c) If destination = source then CF = 0, ZF = 1, SF = 0.

Examples

CMP AL, 0FFH	Compare AL with immediate number FFH.
CMP AX, BX	Compare AX with BX.
CMP CX, COUNT	Compare CX with COUNT.

☞ DAA (Decimal Adjust Accumulator)

- DAA instruction is used to convert the result of the addition of two packed BCD numbers into a packed BCD number.
- DAA only works on AL register. So, DAA instruction must be used after the ADD/ADC instruction.
- The ADD/ADC instruction adds the two BCD number in hexadecimal format and DAA instruction convert this hexadecimal result to BCD result.
- The working of DAA instruction is given below.
  1. If the value of the lower nibble in AL accumulator is greater than 9 or if AF flag is set, the DAA instruction adds 6 to the lower nibble of AL accumulator.
  2. If the value of the higher nibble in AL accumulator is greater than 9 or if CF flag is set, the DAA instruction adds 60 to the higher nibble of AL accumulator.

Flag affected : CF, PF, AF, SF, ZF and OF is undefined.

Operation

(a) If lower nibble of AL > 9 or AF = 1, then AL = AL + 06.

(b) If higher nibble of AL > 9 or CF = , then AL = AL + 60.

(c) If both above condition are satisfied, then AL = AL + 66.

Example

If AL = 99 BCD and BL = 99 BCD

Then ADD AL, BL 1001 1001 = AL = 99 BCD

\+ 1001 1001 = BL = 99 BCD

0011 0010 = AL = 32 H and CF=1, AF=1

Now, in above example after addition, Carry and Auxiliary carry flags are set. So DAA instruction adds 6 to higher as well as lower nibbie of AL register to get correct BCD result i.e. 198 BCD of which 98 in AL and Cy =1 as given below.

After the execution of DAA instruction, the result is

Cy =1 0011 0010 = AL = 32 H and AF = 1

\+ 0110 0110

1001 1000 = AL = 98 in BCD form

☞ DAS (Decimal Adjust after Subtraction)

- DAS instruction is used to convert the result of the previous subtraction of two packed BCD numbers to a packed BCD number.
- DAS instruction only works on AL register.
- So, DAS instruction must be used after the SUB/SBB instruction.
- The SUB/SBB instruction subtracts the two BCD number in hexadecimal format and DAS instruction convert this hexadecimal result to BCD result.
- The working of DAS instruction is given below.
  1. If the value of the lower nibble in AL accumulator is greater than 9 or if AC flag is set, the DAS instruction subtracts 6 to the lower nibble of AL accumulator.
  2. If the value of the higher nibble in AL accumulator is greater than 9 or if CF flag is set, the DAS instruction subtracts 6 to the higher nibble of AL accumulator.

Flag affected: CF, PF, AF, SF, ZF and OF is undefined.

Operation

(a) If lower nibble of AL > 9 or AF = 1 then AL = AL – 06.

(b) If higher nibble of AL > 9 or CF = 1 then AL = AL – 60.

(c) If both above condition is satisfied then AL = AL – 66.

Example

If AL = 55 BCD and BL = 49 BCD

Then SUB AL, BL 0101 0101 = AL = 55 BCD

– 0100 1001 = BL = 49 BCD

0000 1100 = AL = 0C H and Cy = 0, AF = 1

Now, in above example after subtraction, the value of lower nibble of accumulator is greater than 9 as well as AF flag is set. So DAS instruction subtracts 6 from lower nibble of AL register to get correct BCD result i.e. 06 BCD as given below.

After the execution of DAS instruction, the result is Cy = 0

0000 1100 = AL = 0CH; AF = 1 and lower nibble > 9

– 0000 0110

0000 0110 = AL = 06 in BCD form

☞ NEG destination

- This instruction converts the number byte/word in a destination in the 2's complement and store result in the destination which may be a register or a memory location specified by any one of the addressing modes.

Flag affected: OF, CF, PF, AF, SF, ZF.

Operation

Destination ← 2nd Complement of destination

Examples

NEG AX	Replace the number in AX with its 2's complement.
NEG BYTE PTR[BX]	Replace byte at offset [BX] in DS with its 2's complement.

☞ MUL source

- This instruction is used to multiply an **unsigned** byte from source with an **unsigned** byte in the AL register, or an **unsigned** word from source with an **unsigned** word in the AX register.
- The source must be a any register or a memory location.

- When a byte is multiplied with the byte in AL, then the result is stored in AX because the result of multiplication of two 8-bit i.e. bytes' numbers is maximum 16 bits.
- When a word is multiplied with the word in AX, then the MSW of result is stored in DX and LSW of result in AX register because the result of multiplication of two 16-bit numbers is maximum 32-bits.
- If the MSB or MSW of the result is zero, then CF and OF both will be set.

Flag affected : OF, CF and PF, AF, SF, ZF are undefined.

Operation

(a) If source is byte then AX ← AL * unsigned 8 bit source.

(b) If source is word then DX:AX ← AX * unsigned 16 bit source.

Examples

MUL BL	Muitiply AL by BL, result in AX.
MUL CX	Multiply AX by CX, result in DX:AX.
MUL BYTE PTR [BX]	Multiply AL by byte in DS pointed by [BX], result in AX.

☞ IMUL source

- This instruction is used to multiply a **signed** byte from source with a **signed** byte in the AL register during signed byte multiplication and a **signed** word from source with a **signed** word in the AX register during signed word multiplication.
- The source must be a register or a memory location.
- When a byte is multiplied with the byte in AL, then the result is stored in AX because the result of two 8-bit i.e. bytes numbers is maximum 16 bits.
- When a word is multiplied with the word in AX, then the MSB result is stored in DX and LSB in AX register because the result of two 16-bits i.e. words numbers is maximum 32-bits.
- If the magnitude of the product does not requires all the bits of the destination, the unused bits are filled with the copies of the sign bit.

Flag affected : OF, CF and PF, AF, SF, ZF are undefined.

Operation

(a) If source is byte then AX ← AL * signed 8 bit source.

(b) If source is word then DX:AX ← AX * signed 16 bit source.

Examples

IMUL BL	Multiply AL by BL, result in AX.
IMUL CX	Multiply AX by CX, result in DX:AX.
IMUL BYTE PTR [BX]	Multiply AL by byte in DS pointed by [BX], result in AX.

Example of multiplication of signed byte with signed word.

MOV CX, multiplier	Load signed word multiplier in CX.
MOV AL, multiplicand	Load signed byte multiplicand in AL.
CBW	Extend sign of AL into AH.
IMUL CX	Word multiplies, result in DX:AX.

☞ DIV source

- This instruction divides an **unsigned** word by an **unsigned** byte during 16/8 division, and to divide **unsigned** double word i.e. 32-bits by an **unsigned** word during 32/16 division.
- During the division of a word by a byte, the word (dividend) must be in the AX register and a byte (divisor) may in any 8 bit register or memory location.
- After the division operation, 8 bit quotient will be available in AL register and 8 bit remainder will available in AH register
- While dividing double word by a word, the most significant word of the double word should be in DX and the least significant word of the double word should be in AX.
- After the division operation, 16 bit quotient will be available in AX register and 16 bit remainder in DX register
- If word or double word is divided by 0 or the quotient is too large to fit in AL or AX i.e. greater than FFH or FFFFH, the 8086 will automatically generates a type 0 interrupt i.e. divide by 0 interrupt or divide overflow interrupt.
- During the division of double word by word, the dividend must be in DX: AX for double word or AX for word, but source of the divisor should be a word or byte register or a memory location.
- During the division of a byte by a byte, we must first store dividend byte in AL and fill AH with all 0's for unsigned dividend.
- When we want to divide a word by a word, we must first store dividend word in AX and fill DX with all 0's for unsigned dividend.

Flag affected : None and OF, CF, PF, AF, SF, ZF are undefined.

Operation

(a) If source is byte then
AL ← AL / unsigned 8 bit source
AH ← AL **MOD** unsigned 8 bit source.

(b) If source is word then
AX ← DX: AX / unsigned 16 bit source
DX←DX:AX **MOD** unsigned 16 bit source.

Examples

DIV BL	Divide word in AX by byte in BL, quotient in AL and remainder in AH.
DIV CX	Divide double word in DX and AX by word in CX, quotient in AX and remainder in DX.
DIV NUM [BX]	Divide word in AX by byte in memory location pointer by [BX].

☞ IDIV source

- This instruction divides an **signed** word by an **signed** byte during 16/8 division, and to divide **signed** double word i.e. 32-bits by an **signed** word during 32/16 division.
- During the division of a word by a byte, the word (dividend) must be in the AX register and a byte (divisor) may in any 8 bit register or memory location.

- After the division operation, 8 bit quotient will be available in AL register and 8 bit remainder will available in AH register
- During the division of double word by word, the dividend must be in DX: AX for double word or AX for word, but source of the divisor should be a word or byte register or a memory location.
- During the division of a byte by a byte, we must first store dividend byte in AL and fill AH with all 0's for unsigned dividend.
- If word or double word is divided by 0 or the quotient is too large to fit in AL or AX i.e. greater than FFH or FFFFH, the 8086 will automatically generates a type 0 interrupt i.e. divide by 0 interrupt or divide overflow interrupt.
- For division, the dividend (numerator) must always be in AX:DX for word denominator and AX for byte denominator, but source of the divisor (denominator) can be a register or a memory location.
- When we want to divide a byte by a byte, we must first store dividend byte in AL and fill all bits in AH with sign bit of AL using CBW instruction.
- When we want to divide a word by a word, we must first store dividend word in AX and fill all bits in DX with sign bit of AX using CWD instruction.

Flag affected : None and OF, CF, PF, AF, SF, ZF are undefined.

Operation

(a) If source is byte then

AL ← AL / signed 8 bit source

AH ←AL **MOD** signed 8 bit source.

(b) If source is word then

AX ← DX:AX / signed 16 bit source

DX ← DX:AX **MOD** signed 16 bit source.

Examples

IDIV BL	Divide a signed word in AX by a signed byte in BL, quotient in AL and remainder in AH.
IDIV CX	Divide a signed double word in DX and AX by a signed word in CX, quotient in AX and remainder in DX.
IDIV NUM [BX]	Divide a signed word in AX by a signed byte in memory location pointer by [BX].

Example of division of signed byte with signed byte.

MOV BL,	divisor Load signed byte divisor in BL.
MOV AL,	dividend Load signed byte dividend in AL.
CBW	Extend sign of AL into AH.
IDIV BH	Byte division, remainder in AH and quotient in AL.

☞ CBW

- This instruction copies the sign of byte in AL to all the bits in AH.
- AH is then said to be the sign extension of AL.
- The CBW operation must be done before performing division of a signed byte in the AL by another signed byte with IDIV instruction.

Operation

AH ← filled with 8th bit of AL i.e. D_7

Example

AX = 00000000 10011011

CBW convert signed byte in AL to signed word in AX

AX = 11111111 10011011

☞ CWD

- This instruction copies the sign bit of a word in AX to all the bits of the DX register.
- In other word, it extends the sign of AX into all of DX.
- The CWD operation must be done before performing division of a signed word in AX by another signed word with the IDIV instruction.

Operation

DX ← filled with 16th bit of AX i.e. D_{15}.

Example

DX = 00000000 00000000

AX = 11110000 11000111.

CWD Convert signed word in AX to signed double word in DX AX.

Result after the execution of CWD.

DX = 11111111 11111111

AX = 11110000 11000111.

☞ AAA [ASCII adjust after addition]

- This instruction can be used to convert the contents of the AL register to unpacked BCD result.
- The higher nibble of the AL register is filled with zeros.
- This instruction should be executed after the ADD instruction and the result is placed in the AL register.
- The working of AAA instruction is as follows.
- Before the execution of AAA instruction, AH should be loaded with 0.

Flag affected : AF and CF

Operation

1. Clear the high order nibble of AL i.e. AL=AL AND 0F.
2. If lower nibble of AL > 9 or AF = 1 then.
   - (a) AL = AL + 6.
   - (b) AH = AH + 1.
   - (c) AF = CF = 1.
   - (d) AL = AL AND 0FH.

Example

AAA

Before the execution of AAA, Suppose AH = 00H, AL = 0BH

After the execution of AAA AH = 01H, AL = 01H

☞ Difference between AAA and DAA

Sr. No.	AAA	DAA
1.	ASCII adjust after addition instruction converts the contents of the AL register to unpacked BCD result.	DAA Decimal adjust accumulator Instruction Convert Hexadacimal result to BCD result.
2.	Operation : 1. Clear the high order nibble of AL i.e. AL=AL AND 0F. 2. If lower nibble of AL > 9 or AF = 1 then. (a) AL = AL + 6. (b) AH = AH + 1. (c) AF = CF = 1. (d) AL = AL AND 0FH.	Operation : (a) If lower nibble of AL > 9 or AF = 1, then AL = AL + 06. (b) If higher nibble of AL > 9 or CF =1, then AL = AL + 60. (c) If both above condition are satisfied, then AL = AL + 66.

☞ AAS [ASCII Adjust after Subtraction]

- This instruction can be used to convert the contents of the AL register to the BCD result.
- The higher nibble of the AL register is filled with zeros.
- This instruction should be executed after the SUB instruction and the result is placed in the AL register.
- The working of AAS instruction is as follows.

Flag affected AF and CF.

Operation

1. Clear the higher order nibble of AL i.e. AL = AL AND 0F.
2. If lower nibble of AL > 9 or AF = 1 then
   - (a) AL = AL – 6.
   - (b) AH = AH – 1.
   - (c) AF = CF = 1.
   - (d) AL = AL AND 0FH.

Example

AAS

Before the execution of AAS Suppose AH = 02H, AL = 0BH.

After the execution of AAS AH = 01H AL = 05H.

☞ AAM [ASCII Adjust after Multiplication]

- This instruction can be used to convert the result of the multiplication of two valid unpacked BCD numbers.
- This instruction should be issued after the multiplication instruction.
- The operation of MUL on unpacked BCD number is always less than 100; hence, the result will be in the register AL.
- The binary number in AL register is divided by 10 and the quotient is stored in the register AH and remainder is stored in the AL register.
- The working of AAM instruction is as follows :

Flag affected : PF, SF, ZF

Operation :

(a) AL = AL MOD 10.

(b) AH = AL / 10 [Only integer part is considered].

Example

AL = 06	BL = 08.
MUL BL	AX = 30 H (48 in decimal).
AAM	AH = 04 and AL = 08.

☞ AAD [ASCII adjust before division]

- This instruction AAD can be used to convert the unpacked BCD digit in AH and AL registers to the equivalent binary number in the AL register.
- This instruction should be issued before division instruction.
- The division instruction will place the quotient in the AL register and remainder in the AH register.
- The higher nibble of AH and AL are filled with zeros.

Flag affected : PF, SF, ZF

Operation

(a) AL = AH * 10 + AL (b) AH = 00

Example

AX = 19H (25 in decimal) and BL = 07 H (7 in decimal) AAD

DIV BL AL = 03 quotient and AH = 04 remainder

☞ Difference between ADD and ADC

- ADD : The ADD instruction adds a number from some source to a number from some destination without carry.
- ADC : The ADD instruction adds a number from some source to a number from some destination with carry.

Syllabus Topic : Logical or Bit Manipulation Instructions

3.4.3 Logical or Bit Manipulation Instructions

➔ **(MSBTE - S-14, W-14, S-15, W-16, W-17, S-18)**

Q. 3.4.18 Compare the following instructions (2 points) : AND and TEST. (Ref. sec. 3.4.3) S-14, W-16, 2 Marks

Q. 3.4.19 Explain any one logical instruction of 8086 with example. (Ref. sec. 3.4.3) S-14, 2 Marks

Q. 3.4.20 Write any two logical instructions with their functions. Give the syntax with one example each. (Ref. sec. 3.4.3) W-14, 2 Marks

Q. 3.4.21 Differentiate between instructions : ROL and RCL (Ref. sec. 3.4.3) S-15, S-18, 1 Mark

Q. 3.4.22 Explain any four rotation instructions with example. (Ref. sec.3.4.3) W-16, 4 Marks

Q. 3.4.23 Describe any two-bit manipulation instructions. (Ref. sec. 3.4.3) W-17, 4 Marks

Q. 3.4.24 With suitable example, explain following instruction : AND. (Ref. sec. 3.4.3) S-18, 1 Mark

AND destination, source

- This instruction AND's bit-by-bit the source operand with destination operand and result is stored in the destination specified in the instruction.
- The result of each bit position will follow the truth table for a two input AND gate.
- The source operand can be an immediate number, the content of a register, or the content of memory location specified by any one of the 24 ways given in Table 3.2.2.
- The destination can be a register or a memory location.
- The source and destination cannot both be memory locations in the same instruction.

Flag affected : CF = 0, OF = 0, PF, SF, ZF.

Operation :

Destination ← destination **AND** source.

Examples

AND BH, CL	AND byte in CL with byte in BH, result in BH.
AND BX, 00FFH	AND word in BX with immediate cdata 00FFH.
AND CX, [SI]	AND word at offset [SI] in data segment with word in CX. Result in CX register.

OR destination, source

- This instruction OR's each bit in a source byte or word with the corresponding bits in the destination byte or word and the result is stored in destination.
- The result of each bit will follow the truth table of two inputs OR gate.
- The source operand can be an immediate data, the contents of register, or the contents of a memory location specified by any one of 24 ways given in Table 3.2.2.
- The destination can be a register or a memory location.
- The source and destination cannot both be a memory location in the same instruction.

Flag affected : CF = 0, OF = 0, PF, SF, ZF

Operation

Destination ← destination **OR** source

Examples :

OR BH, CL	OR byte in CL with byte in BH, result in BH.
OR BX,	00FFH OR word in BX with immediate data 00FFH. Result in BX.
OR CX, [SI]	OR word at offset [SI] in data segment with word in CX. Result in CX register.

NOT destination

- This instruction inverts each bit of the byte or word at the specified destination i.e. 1's complement.
- The destination can be a register or a memory location specified by any one of 24 ways given in Table 3.2.2.

Flag affected : None

Operation

Destination ← **NOT** destination

Examples

NOT BX	Complement the contents of BX register.
NOT [4000H]	Complement the contents of the memory location 4000H.
NOT BYTE PTR [BX]	Complement the contents of the memory location pointer by [BX].

XOR destination, source

- This instruction perform the logical operation i.e. Exclusive ORs of each bit in a source byte or word with the same number bit in a destination byte or word and stores result in the destination.
- The result for each bit position will be as per the truth table of two inputs XOR gate.
- The source operand may be an immediate number, register, or memory location.
- The destination may be register or a memory location.
- But, the source or destination should not both be memory locations in the same instruction.

Flag affected : CF = 0, OF = 0, PF, SF, and ZF.

Operation

Destination ← destination **XOR** source.

Examples

XOR BH, CL	XOR byte in CL with byte in BH, result in BH.
XOR BX,	00FFHXOR word in BX with immediate data 00FFH.
XOR CX, [SI]	XOR word at offset [SI] in data segment with word in CX. Result in CX register.

TEST destination, source

- This instruction ANDs the contents of a source byte or word with the contents of specified destination byte or word and flags are updated, but neither operands are changed.
- The TEST instruction is often used to set flags before a conditional jump instruction.
- The source operand may be an immediate number, the register, or the memory location.

- The destination operand must be a 8 or 16 register or a memory location.

Flag affected : CF, OF, PF, SF, and ZF.

Operation

Flags ← set for result of (destination **AND** source).

Examples

TEST BH, CL	AND byte in CL with byte in BH, no result, Update PF, SF, ZF.
TEST BX, 00FFH	AND word in BX with immediate data 00FFH, no result, Update PF, SF, ZF.
TEST CX, [SI]	AND word at offset [SI] in data segment with word in CX, no Result, Update PF, SF, ZF.

Difference between AND and TEST

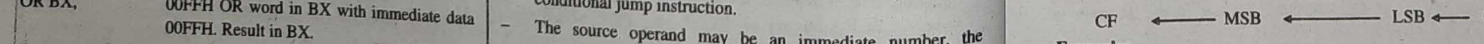
Sr. No.	AND	TEST
1.	Destructive AND instruction means destination is modified after the execution of Instructions	Non Destructive AND instruction means destination is not modified after the execution of Instructions
2.	Flag affected : CF = 0, OF = 0, PF, SF, ZF.	Flag affected : CF, OF, PF, SF, and ZF.
3.	Operation : Destination ← destination AND source.	Operation : Flags ← set for result of (destination AND source).

SHL / SAL destination, count

- SHL and SAL are two mnemonics for the same operation.
- This instruction shifts each bit in the specified destination counts times toward left.
- As a bit is shifted out of the LSB position, a 0 is inserted in the LSB position and the MSB will be shifted into the CF.
- In the case of multiple shifts, CF will contain the bit most recently shifted in from the MSB and bits shifted into CF previously will be lost.
- The destination operand can be a byte or a word in a register or in a memory location.
- If the desired number of shifts is one, this can be done by putting a 1 in the count position of the instruction.
- For shifts more than 1 bit position the desired number, then the count value i.e. shift count must be loaded in CL register and put CL register in the count position in the instruction.
- The SAL and SHL instruction can be used to multiply an unsigned binary number by a power of two.

Flag affected : CF, OF, PF, SF, ZP; and AF is undefined.

Operation



Examples

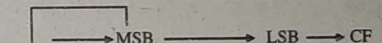
1. CF = 0 BX = 11100101 11010011.
   SAL BX, 1 Shift the contents of BX register by one toward left.
   CF = 1 BX = 11001011 10100110.
2. MOVCL, 04H Load desired number of shifts in CL.
   SAL AX, CL Shift word in AX left CL bits times.

SAR destination, count

- This instruction shifts each bit in the specified destination count times toward right.
- As the shifted out of MSB position, a copy of the old MSB is kept in the MSB position i.e. the sign bit is copied into the MSB.
- The LSB will be shifted into CF.
- In case of multiple shifts; CF will contain the bit most recently shifted in from the LSB, so bits shifted into CF previously will be lost.
- The destination operand can be a byte or word in a register or in a memory location.
- If the desired number of shifts is one, this can be done by putting a 1 in the count position of the instruction.
- For shifts more than 1 bit position the desired number, then the count value i.e. shift count must be loaded in CL register and put CL register in the count position in the instruction.
- The SAR instruction can be used to divide an unsigned binary number by a power of two.

Flag affected : CF, OF, PF, SF, ZP and AF is undefined.

Operation



Examples

1. CF = 0 BX = 11100101 11010011.
   SAR BX, 1 Shift the contents of BX register by one toward left.
   CF = 1 BX = 11110010 11101001.
2. MOV CL,04H Load desired number of shifts in CL.
   SAR AX, CL Shift word in AX right CL bits times.

SHR destination, count

- This instruction shifts each bit in the specified destination counts times to right.
- As a bit is shifted right out of the MSB position, a 0 is inserted in its place.
- The bit shifted out of the LSB position goes to the CF.
- In the case of multiple shifts, CF will contain the bit most recently shifted in from the LSB.
- Bits shifted into CF previously will be lost.
- The destination operand can be a byte or a word in a register or in a memory location specified by any one of 24 ways specified in Table 3.2.2.

- If the desired number of shifts is one, this can be done by writing 1 in the place of count.
- But, for multiple shifts, the number of desired shifts is loaded into the CL register and CL is kept in the place of count.
- The SHR instruction can be used to divide an unsigned binary number by a power of two.

Flag affected : CF, OF, PF, SF, ZF and AF is undefined.

Operation

0 ⟶ MSB ⟶ LSB ⟶ CF

Examples :

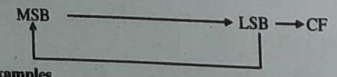
1.	CF = 0	BX = 11100101 11010011.
	SHR BX, 1	Shift the contents of BX register by one toward left.
	CF = 1	BX = 01110010 11101001.
2.	MOV CL, 04H	Load desired number of shifts in CL.
	SHR AX, CL	Shift word in AX right CL bits times.

☞ ROR destination, count [Rotate right without carry]

- This instruction is used rotates all the bits of the specified byte or word by count times toward right.
- The bit rotated out of LSB is goes into the MSB and also copied to CF.
- In the case of multiple bit rotates the CF will contain a copy of the bit most recently moved out of LSB.
- The destination operand may be a byte or word in a register or in a memory location.
- If the desired number of rotation is one, this can be done by writing 1 in the place of count.
- But, for multiple rotations, the number of desired rotation is loaded into the CL register and CL is kept in the place of count.
- This instruction can be used to swap the nibbles in a byte or to swap the bytes in a word.
- It can also be used to rotate a bit into the CF where it can be checked and acted upon by the conditional jump instruction, JC [jump if carry] or JNC [jump if no carry].

Flag affected : OF, CF

Operation



Examples

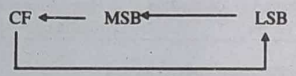
1.	CF = 0	BL = 0011 1011.
	ROR BL, 1	Rotate all bits in BL right by one bit position.
	CF = 1	BL = 1001 1101.
2.	MOV CL, 08H	Load count in CL.
	CF = 1	AX = 10011110 00111000.
	ROR AX, CL	Rotate all bits in AX right by 8 bit position i.e. Swapping of byte.
	CF = 0	AX = 00111000 10011110.

☞ ROL destination, count [Rotate left without carry]

- This instruction rotates all of the bits of the specified byte or word count times toward left.
- The bit moved out of MSB is rotated around into the LSB and also copied to CF.
- In the case of multiple bit rotates the CF will contain a copy of the bit most recently moved out of MSB.
- The destination operand can be a byte or word in a register or in a memory location specified by one of the 24 ways in Table 3.2.2.
- If the desired number of rotation is one, this can be done by writing 1 in the place of count.
- But, for multiple rotations, the number of desired rotation is loaded into the CL register and CL is kept in the place of count.
- This instruction can be used to swap the nibbles in a byte or to swap the bytes in a word.
- This instruction can also be used to rotate a bit into the CF and then it can be checked and acted upon using the conditional jump instruction, JC [jump if carry] or JNC [jump if no carry].

Flag affected : OF, CF

Operation



Examples

1.	CF = 0	BL = 1011 1010.
	ROL BL, 1	Rotate all bits in BL left by one bit position.
	CF = 1	BL = 0111 0101.
2.	MOV CL, 08H	Load count in CL.
	CF = 1	AX = 00011110 00111001.
	ROL AX, CL	Rotate all bits in AX left by 8 bit position i.e. swapping of byte.
	CF = 0	AX = 00111001 00011110.

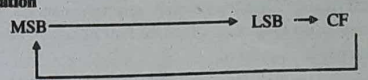
☞ RCR destination, count [Rotate right with carry]

- This instruction is used to rotate all the bits in a specified byte or word with carry by counts times towards right i.e. the LSB of the operand is rotated into the carry flag and the bit in carry flag is rotated into the MSB of the operand.
- In the case of multiple bit rotates the CF will contain a copy of the bit most recently moved out of LSB.
- The destination operand can be a byte or word in a register or in a memory location specified by one of the 24 ways in Table 3.2.2
- If the desired number of rotation is one, this can be done by writing 1 in the place of count.

- But, for multiple rotations, the number of desired rotation is loaded into the CL register and CL is kept in the place of count.

Flag affected OF, CF

Operation



Examples

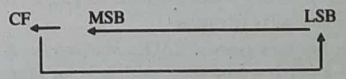
1.	CF = 0	BL = 0011 1011.
	RCR BL, 1	Rotate all bits in BL right by one bit position.
	CF = 1	BL = 0001 1101.
2.	MOV CL, 08H	Load count in CL.
	CF = 1	AX = 10011110 00111000.
	RCR AX, CL	Rotate all bits in AX right by 8 bit position i.e. Swapping of byte.
	CF = 0	AX = 01110001 10011110.

☞ RCL destination, count [Rotate left with carry]

- This instruction is used to rotate all the bits in a specified byte or word with carry by counts times towards left i.e. the MSB of the operand is rotated into the carry flag and the bit in carry flag is rotated into the LSB of the operand.
- In the case of multiple bit rotates the CF will contain a copy of the bit most recently moved out of MSB.
- The destination operand can be a byte or word in a register or in a memory location specified by one of the 24 ways in Table 3.2.2.
- If the desired number of rotation is one, this can be done by writing 1 in the place of count.
- But, for multiple rotations, the number of desired rotation is loaded into the CL register and CL is kept in the place of count.

Flag affected : OF, CF

Operation



Examples

1.	CF = 1	BL = 0011 1011.
	RCL BL, 1	Rotate all bits in BL left by one bit position.
	CF = 0	BL = 0111 0111.
2.	MOV CL,	08H Load count in CL.
	CF = 1	AX = 10011110 00111000.
	RCL AX, CL	Rotate all bits in AX left by 8 bit position i.e. Swapping of byte.
	CF = 0	AX = 00111000 11001111.

☞ Differentiate between RCR and RCL

Q. 3.4.25 Differentiate RCR and RCL instructions on basis of their operation, with example. (Ref. sec. 3.4.3)

Sr. No.	RCR	RCL
1.	Use to rotate all the bits in specified byte or word with carry by counts times to right.	Use to rotate all the bits in specified byte or word with carry by counts times to left.
2.	LSB of operand goes to CF and CF bit goes to MSB of operand.	MSB of operand goes to CF and CF bit goes to LSB of operand.
3.	In case of multiple bits rotation, CF will contains a copy of the bit most recently moved out of LSB.	In case of multiple bits rotation, CF will contains a copy of the bit most recently moved out of MSB.
4.	Example: If CF = 0, BL = 0011 1011. RCR BL, 1 Then CF = 1, BL = 0001 1101.	Example: If CF = 0 BL = 0011 1011. RCL BL, 1 Then CF = 0 BL = 0111 0110.

☞ Difference between ROL and RCL

- **ROL :** This instruction rotates all of the bits of the specified byte or word count times toward left without carry.
- **RCL :** This instruction rotates all of the bits of the specified byte or word count times toward left with carry.

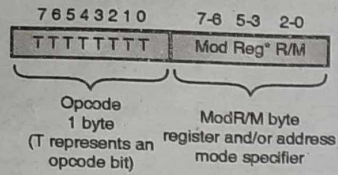
Q. 3.4.26 The shift and rotates instructions are the same accepts for 3 bits. Which distinguish these instructions from each other ? (Ref. sec. 3.4.3)

The op-code format of the entire shift and rotate instructions are given below. The entire shift and rotate instructions are of two bytes. Out of these two op-code bytes, first byte of the op-code of entire shift and rotate instructions are same.

SHL/SAL = shift Logical Arithmetic Left
SHR = Shift Logical Right
SAR = Shift Arithmetic Right
ROL = Rotate Left
ROR = Rotate Right
RCL = Rotate Through Carry Flag Left
RCR = Rotate Through Carry Right

Instruction code	
7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
1 1 0 1 0 0 v w	mod 1 0 0 r/m
1 1 0 1 0 0 v w	mod 1 0 1 r/m
1 1 0 1 0 0 v w	mod 1 1 1 r/m
1 1 0 1 0 0 v w	mod 0 0 0 r/m
1 1 0 1 0 0 v w	mod 0 0 1 r/m
1 1 0 1 0 0 v w	mod 0 1 0 r/m
1 1 0 1 0 0 v w	mod 0 1 1 r/m
Byte 1	Byte 2

Now, compare above op-code formats of entire shift and rotate instructions with the following general op-code format of the instruction.



In second byte, the bits 3,4,5 are different depending on the operation specified by the instructions and remaining bits i.e. 0,1,2,6,7 are same for entire shift and rotate instruction. The Reg field specifies general purpose register operands but for entire shift and rotate instructions Reg field is used as op-code extension field i.e. TTT. These op-code extension field TTT is different in entire shift and rotate instruction. These op-code extension field TTT differentiate between the entire shift and rotate instructions.

Syllabus Topic : Program Control Transfer or Branching Instructions

3.4.4 Program Control Transfer or Branching Instructions

➔ **(MSBTE - W-14, W-15, S-16, S-17, W-17)**

Q. 3.4.27 What are the functions of CALL and RET instruction? Write syntax of CALL and RET instructions. (Ref. sec. 3.4.4) W-14, W-15, S-16, 4 Marks

Q. 3.4.28 Name the different types of jump instructions used in 8086 assembly language program. (any eight) (Ref. sec. 3.4.4) W-15, 4 Marks

Q. 3.4.29 Explain the following instructions of 8086 with suitable example.
(i) LOOP (ii) INTO
(Ref. sec. 3.4.4) S-16, 4 Marks

Q. 3.4.30 Write any two conditional and two unconditional branching instruction with their function. Give the syntax with one example each.
(Ref. sec. 3.4.4) S-17, 4 Marks

Q. 3.4.31 Explain following instructions : LOOP.
(Ref. sec. 3.4.4) W-17, 2 Marks

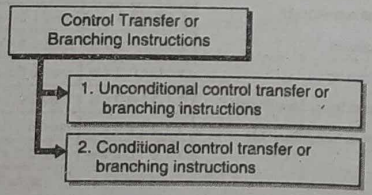


Fig. 3.4.2 : Control Transfer or Branching Instructions

- The control transfer or branching instructions are used to change the path or flow of execution of the program to new address specified in the instruction directly or indirectly.
- When this type of instruction is executed, the contents of CS and IP registers get loaded with new values of CS and IP corresponding to location where the flow or path of execution is going to be transferred.
- Depending on the addressing modes specified in Table 3.2.2, the CS may or may not modify.
- There are two types of control transfer or branching instructions as given as follows.

➔ **(1) Unconditional control transfer or branching instructions**

- This type of instruction transfers the control of execution to the specified memory location independent of any condition or status.
- The CS and IP are unconditionally modified with a new CS and IP values.

➔ **(2) Conditional control transfer or branching instructions**

- This type of instruction transfers the control of execution to the specified memory location dependant of any condition or status provided by the result of the previous operation which satisfies a particular condition i.e. status of flags, otherwise, the execution continues in normal flow sequence.
- In other word, using this type of instruction the control will be transferred to a particular memory location, if a particular flag satisfies the condition.

1. Unconditional control transfer or branching instructions

☞ **CALL a procedure**

- The CALL instruction is used to transfer the program control to the sub-program or subroutine.
- There are two basic types of CALLs, *near* and *far*. A near CALL is a call to a procedure, which is in the same code segment as the CALL instruction.
- When the 8086 executes the near CALL instruction, the stack pointer is decrement by two and copies the offset i.e. IP of the next instruction after the CALL instruction on the stack.
- This offset value is used to transfer back the program control to the calling program after the execution of subroutine or procedure.
- Then, the 8086 loads the offset of first instruction of the procedure into the IP and RET instruction at the end of the procedure will return execution to the instruction after the CALL instruction in the calling program by copying the offset value stored back to IP.

- A far CALL is used to call a procedure, which resides in another code segment from that which contains the CALL instruction.
- When CPU 8086 executes the far CALL instruction, the stack pointer is decrement by two and copies the contents of the CS register onto the stack.
- Again, the stack pointer is decrement by two and copies the contents of IP i.e. offset of the next instruction after the CALL instruction onto the stack.
- Finally, it loads CS with the segment base address of the code segment that contains the procedure and IP with the offset of the first instruction of the procedure in that segment.
- A RET instruction at the end of the procedure will return execution to the next instruction of the calling program by restoring the saved value of CS and IP from the stack.
- Operation

1.	If NEAR CALL,	then SP ← SP – 2 Save IP on stack IP ← address of procedure
2.	If FAR CALL	then SP ← SP — 2 SP ← CS i.e. Save CS on stack CS ← New segment base address of the called procedure SP ← SP – 2 SP ← IP i.e. Save IP on stack IP ← New offset address of the called procedure

Examples

CALL DELAY	Direct within the segment that calls the procedure of name DELAY.
CALL BX	Indirect within the segment where BX contains the offset of the first Instruction of the procedure and replace the contents of IP with contents of BX register.
CALL FAR PTR SHOW	Direct to another segment i.e. far or inter-segment; SHOW is the name of procedure and must be declared FAR with SHOW PROC FAR at its start. The assembler will determine the code segment base for the segment, which contains the procedure and the offset of the start of the procedure in that segment.

☞ **RET instruction**

- The RET instruction is used to return the program execution control from a procedure to the next instruction immediate after the CALL instruction in the calling program.
- If the procedure is a near procedure, then the return will be done by restoring the value of IP with a word from the stack top.
- The word from the stack top is offset of the next instruction after the CALL instruction in the calling program that was pushed on the stack by the CALL instruction.
- So, the stack pointer is incremented by two and return address is popped from the stack to IP.
- If the procedure is far procedure i.e. in a different code segment from the CALL instruction which calls it, the IP will be replaced by the word from stack top which is nothing but the offset of the next instruction after the CALL instruction of the calling program pushed by the by the CALL instruction.
- Again, the stack pointer is incremented by two; the CS is replaced with the current stack top, which is the segment base of the segment where CALL instruction resides.
- After the replacement of CS, again the stack pointer is incremented by two.

Operation

1.	For NEAR Return then	IP ← content of top of stack. SP ← SP + 2.
2.	For FAR Return then	IP ← contents of top of stack. SP ← SP + 2. CS ← contents of top of stack. SP ← SP + 2.

☞ **INT N [N = type of interrupt]**

- This instruction causes the 8086 to call a far procedure in a manner similar to the way in which the 8086 respond to an interrupt signal on its INTR or NMI inputs.
- The term type in the instruction refers to the number between 0 and 255 that identifies the interrupt.
- When the 8086 execute an INT type instruction, it will perform following operation.
  (a) Decrement the stack pointer by two and push the flags on the stack.
  (b) Decrement the stack pointer by two again and push the contents of the CS.
  (c) Decrement the stack pointer by two again and push the offset of the next instruction after the INT type instruction on the stack.
  (d) Get the new value for IP from an absolute memory address of 4 times the type specified in the instruction. For example, for an INT 8 instruction, the new IP will be read from address 00020H.
  (e) Get a new value for CS from an absolute memory address of 4 times the type specified in the instruction plus 2.
  (f) Reset both the IF and the TF flags, other flags are not affected by this instruction.

☞ **INTO Instruction [Interrupt on overflow]**

- If the overflow flag OF is set, this instruction will generate type 4 interrupt and causes the 8086 to do an indirect far call a procedure which is written by the user to handle the overflow condition.
- When the 8086 executes an INTO instruction, it will perform following operation.
  (a) Decrement the stack pointer by two and push the flags on the stack.

(b) Decrement the stack pointer by two and push CS on the stack.

(c) Decrement the stack pointer by two again and push IP contains the offset of the next instruction after an INTO instruction on the stack.

(d) Get the new value for IP from an absolute memory address 00010H in IVT.

(e) Get a new value for CS from an absolute memory address 00012H in IVT.

(f) Reset the TF and the IF, other flags are not affected by this instruction.

IRET instruction

- The IRET instruction is used at the end of the interrupt service procedure to return the execution to the interrupted program.
- During the execution of this instruction, the 8086 copies the saved value of IP from the stack to IP, the saved value of CS from the stack to CS and saved value of flags back to the flag register.
- Functions performed by this instruction are :
  (a) IP is popped from the stack then SP ← SP + 2.
  (b) CS is popped from the stack then SP ← SP + 2.
  (c) Flag register is popped from the stack then SP ← SP + 2.

JMP destination address [Jump unconditional]

- This instruction unconditionally transfers the control of execution to the specified address using an 16-bit displacement or **CS:IP**.
- No flags are affected or checked by this instruction.
- If the target of JMP is in the same code segment, it requires only the instruction pointer IP to be changed to transfer control to the target location.
- Such a jump is called as intra-segment jump or *near* jump.
- If the target for the instruction JMP is in different code segment from that containing the instruction JMP, then IP and CS will be changed to transfer control to the target location. Such a jump is called as *far* or inter-segment jump.

Difference Between Inter and Intra Segment Jump

Sr. No.	Inter-Segment Jump	Intra-Segment Jump
1.	*Intersegment jumps* can transfer control to a instructions in a *different* code segment.	*Intrasegment jumps* are always *between* instructions in the same code segment.
2.	Intersegment jumps is called as FAR jump	Intra Segment jump is called as NEAR jump
3.	Requires Code segment register CS and instruction pointer IP to be changed to transfer control to the target location.	Requires only the instruction pointer IP to be changed to transfer control to the target location.
4.	Example : jmp adrs32 ;direct intersegment	Example : Jmp disp16 ;direct intrasegment

Examples

JMP DOWN
JMP FAR PTR SKIP

Conditional jump instructions

- The conditional JMP instruction transfer the control to the target location if some specified condition is satisfied.
- Conditional JMP instructions are normally used after compare instruction or arithmetic or logical instructions.
- The different types of conditional instruction are given in Table 3.4.1.

Table 3.4.1

Instruction	Function	Syntax
JCXZ	Jump if CX register is Zero	JCXZ label
LOOP	CX = CX – 1, jump if CX ≠ Zero	LOOP label
LOOPE/ LOOPZ	CX = CX – 1, jump if CX ≠ Zero and ZF = 1	LOOPE label LOOPZ label
LOOPNE/ LOOPNZ	CX = CX – 1, jump if CX ≠ Zero and ZF = 1	LOOPNE label LOOPNZ label
JA	Jump if above [CF = 0 and ZF = 0]	JA label
JNBE	Jump if not below or equal [CF = 0 and ZF = 0]	JNBE label
JAE/ JNB/ JNC	Jump if above or equal [CF = 0] Jump if not below [CF = 0] Jump if no carry [CF = 0]	JAE label JNB label JNC label
JB	Jump if below [CF = 1]	JB label
JNAE/ JC	Jump if not above or equal [CF = 1] Jump if carry [CF = 1]	JNAE label JC label
JE/ JZ	Jump if equal [ZF = 1] Jump if Zero [ZF = 1]	JE label JZ label
JG	Jump if greater SF = OF and ZF = 0 after signed mathematics	JG label
JNLE	Jump if not less or equal SF = OF and ZF = 0 after signed mathematics.	JNLE label

Instruction	Function	Syntax
JGE/ JNL	Jump if greater or equal SF = OF after signed math Jump if not less SF = OF after signed math	JGE label JNL label
JNGE / JL	Jump if not greater or equal SF≠ OF after signed math Jump if less than SF ≠ OF after signed math	JNGE label JL label
JLE JNG	Jump if less than or equal SF ≠ OF and ZF = 1 After signed math Jump if not greater SF ≠ OF and ZF = 1 after signed math	JLE label JNG label
JNE/ JNZ	Jump if not equal ZF = 0 Jump if not Zero ZF = 0	JNE label JNZ label
JNO	Jump if not overflow OF = 0	JNO label
JNP JPO	Jump if not parity PF = 0 Jump if Parity ODD PF = 0	JNP label JPO label
JNS	Jump if not sign SF = 0	JNS label
JO	Jump if Overflow OF = 1	JO label
JP JPE	Jump if parity EVEN PF = 1 Jump if Parity equal PF = 1	JP label JPE label
JS	Jump if sign SF = 1	JS label

3.4.4(A) Comparison of Jump and Call Instruction in 8086

➔ **(MSBTE - S-15, S-18)**

Q. 3.4.32 Compare between Jump and Call instruction in 8086. (Ref. sec. 3.4.4(A)) S-15, S-18, 4 Marks

Sr. No.	JMP Instruction	CALL instruction
1.	A JMP instruction permanently changes the IP for near jmp and CS:IP for far jmp.	A CALL instruction store IP for near call and CS:IP for far call on the stack so that the original program execution sequence can be resumed.
2.	Does not requires RET instruction.	RET instruction is required to return to calling program
3.	Conditional JMP instructions are available which transfer program control to the target location if some specified condition is satisfied.	No such Conditional CALL instructions are available.
4.	Stack is not used by JMP instruction	Stack is used by CALL instruction.

3.4.4(B) Comparison of JNC and JMP Instructions in 8086

➔ **(MSBTE - S-15, S-18)**

Q. 3.4.33 Write difference between the following instructions JNC 2000H and JMP 2000H (Ref. sec. 3.4.4(B)) S-15, 4 Marks

Q. 3.4.34 Differentiate between the following instructions : JMP and JNC. (Ref. sec. 3.4.4(B)) S-15, 4 Marks, S-18, 1 Mark

Sr. No.	JNC 2000H	JMP 2000H
1.	Conditional branching instruction	Unconditional branching
2.	CF flags checked by this instruction but CF flag is unaffected	No flags are affected or checked by this instruction
3.	If CF=0 (reset), the program control transfer to memory location 2000H	The program control is transfer to 2000H without checking the condition of any flags
4.	Conditional JNC instruction is normally used after compare instruction or arithmetic or logical instructions.	Unconditional JMP instruction is used when program control to be transfer any part of the program after any instruction

Difference between JMP and JNC

- JMP : The program control is transfer to 2000H without checking the condition of any flags
- JNC : If CF=0 (reset), the program control transfer to the specified memory location.

Syllabus Topic : Process Control Instructions

3.4.5 Process (Machine) Control Instructions

➔ **(MSBTE - W-16, S-17)**

Q. 3.4.35 List four machine control instruction and state their function. (Ref. sec. 3.4.5) W-16, 4 Marks

Q. 3.4.36 List and explain any four process control instruction with their function. (Ref. sec. 3.4.5) S-17, 4 Marks

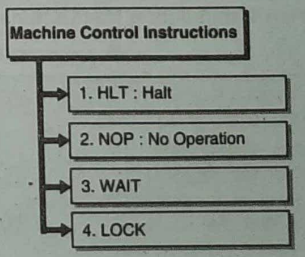


Fig. 3.4.3 : Machine Control Instructions

➜ 1. HLT : Halt

- The instruction HLT causes the processor to enter the halt state.
- The CPU stops fetching and executing of the instruction.
- The CPU can be brought out of the halt state with the occurrence of any one of the following events.
  1. Interrupt signal on INTR pin.
  2. Interrupt signal on NMI pin.
  3. Reset signal on RESET pin.

➜ 2. NOP : No Operation

- This instruction is used to add wait state of three-clock cycles and during these three clock cycles CPU dose not perform any operation.
- This instruction can be used to add delay loop in the program and delaying the operation before proceeding to read or write from the port.

➜ 3. WAIT

The instruction WAIT causes processor to enter into an ideal state or a wait state and continues to remain in that the processor receives state until one of the following signal.

1. Signal on processor TEST pin.
2. A valid interrupt on INTR pin.
3. A valid interrupt on NMI pin.

This signal is used to synchronize with other external hardware such as math co-processor 8087.

➜ 4. LOCK

- This instruction prevents other processors to take the control of shared resources.
- In multiprocessor system, the individual processors have their own local buses and memory and then processor are connected together by a system bus to access the shared system resources such as disk drives or memory or DMA.
- The LOCK instruction is used as a prefix to the critical instruction that has to execute.

Example

LOCK IN AL, 80H

3.4.6 Flag Manipulation Instructions

➜ **(MSBTE - W-16, S-18)**

Q. 3.4.37 State the function of following instruction of 8086.
(i) STC (ii) CMC
(iii) CLD (iv) STI
(Ref. sec. 3.4.6) **W-16, 4 Marks**

Q. 3.4.38 State the function of STC and CMC instructions of 8086. (Ref. sec. 3.4.6) **S-18, 4 Marks**

This type of instruction are used to changed the status of flags in the flag register such as CF, DF, IF.

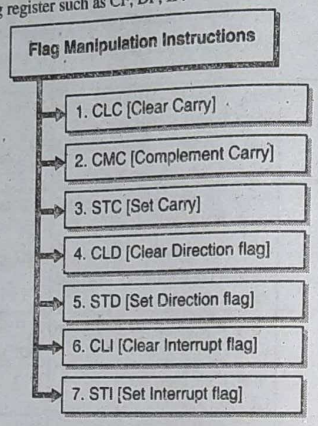


Fig. 3.4.4 : Flag Manipulation Instructions

➜ 1. CLC [Clear Carry]

This instruction clears the carry flag. CF ← 0.

➜ 2. CMC [Complement Carry]

This instruction complements the carry flag.

CF ← ~ CF.

➜ 3. STC [Set Carry]

This instruction set the carry flag. CF ← 1.

➜ 4. CLD [Clear Direction flag]

This instruction clears the direction flag. DF ← 0.

➜ 5. STD [Set Direction flag]

This instruction set the direction flag. DF ← 1.

➜ 6. CLI [Clear Interrupt flag]

This instruction clears interrupt flag. IF ← 0.

➜ 7. STI [Set Interrupt flag]

This instruction set the interrupt flag. IF ← 1.

Syllabus Topic : String Operation Instruction

3.4.7 String Manipulation Instructions

➜ **(MSBTE - S-14, W-14, S-15, W-15, S-16, W-17, S-18)**

Q. 3.4.39 Explain following string instructions and respective Prefix
(a) REP MOVSW (b) REP CMPSB
(Ref. sec. 3.4.7) **S-14, 4 Marks**

Q. 3.4.40 List and explain any four string operation instructions with their functions and syntax.
(Ref. sec. 3.4.7) **W-14, 4 Marks**

Q. 3.4.41 Explain the instruction of 8086 microprocessor with their syntax : STRCMP.
(Ref. sec. 3.4.7) **W-14, 2 Marks**

Q. 3.4.42 Describe various string instructions in brief.
(Ref. sec. 3.4.7) **S-15, S-18, 4 Marks**

Q. 3.4.43 List the string related instructions of 8086 microprocessor and explain any two instructions.
(Ref. sec. 3.4.7) **W-15, 4 Marks**

Q. 3.4.44 Explain any two string operation instructions with suitable example.
(Ref. sec. 3.4.7) **S-16, 4 Marks**

Q. 3.4.45 Describe any two-string operation instruction of 8086 with syntax & one example of each.
(Ref. sec. 3.4.7) **W-17, 4 Marks**

A string is contiguous block of bytes or words and can used to hold any type of data or information that will fit into bytes or words. There are number of operation performed with string. The 8086 microprocessor supports string instruction for string movement, scan, comparison, load and store.

MOVS : Move String.
MOVSB : Move String Byte.
MOVSW : Move String Word.

- The instruction MOVS transfers a byte or a word from the source string to the destination string.
- The source must be in the data segment and destination must be in extra segment.
- The offset of the source byte or a word must be placed in SI register, which is represented as DS:SI and offset of the destination byte or a word must be in DI register, which is represented as ES : DI.
- On the execution of this instruction, SI and DI register are automatically incremented by one to point next element of source and destination.
- If the direction flag is reset [DF=0], the register SI and DI will be incremented by *one* for **byte move** and incremented by *two* for **word move**.
- If the direction flag is set [DF=1], the register SI and DI will be decremented by *one* for **byte move** and decremented by *two* for **word move**.
- The DS:SI and ES:DI register must be loaded prior to the execution of MOVS instruction.
- Another way to move a byte or word string is by using implicit instruction MOVSB and MOVSW.
- The instruction MOVSB is used to transfer a byte from source to destination and the instruction MOVSW is used to transfer a word from source to destination.
- In multiple byte or word moves, the count must be loaded in CX register which functions as a counter.

Operation

ES:[DI] ← DS:[SI].

If byte movement
For DF = 0 SI ← SI + 1 and DI ← DI + 1.
For DF = 1 SI ← SI – 1 and DI ← DI – 1.

If word movement
For DF = 0 SI ← SI + 2 and DI ← DI + 2.
For DF = 1 SI ← SI – 2 and DI ← DI – 2.

Examples

```
MOV AX, @data
MOV DS, AX
MOV ES, AX
CLD
MOV SI, OFFSET S_STRING
MOV DI, OFFSET D_STRING
```

A] MOVS S_STRING, D_STRING
B] MOVSB
C] MOVSW

LODS : Load String.
LODSB : Load String Byte.
LODSW : Load String Word.

- The instruction LODS transfer a byte or a word from the source string pointed by SI in DS to AL for byte or AX for word.
- On execution of a string instruction, SI is automatically updated to point next element of the source string.
- If DF = 0, the register SI will be incremented by 1 for byte and incremented by 2 for word.
- If DF = 1, the register SI will be decremented by 1 for byte and decremented by 2 for word.
- In the instruction LODS, the source must be explicitly declared either with DB or with DW.
- Another way to load a byte or word string is by using implicit instruction LODSB and LODSW.
- The instruction LODSB is used to transfer a string byte from source to AL and the instruction LODSW is used to transfer a string word from source to AX.
- In multiple byte or word loads, the count must be loaded in CX register which functions as a counter.

Operation

If byte movement
AL ← DS:[SI]
For DF = 0 SI ← SI + 1
For DF = 1 SI ← SI – 1
If word movement
AX ← DS:[SI]
For DF = 0 SI ← SI + 2
For DF = 1 SI ← SI – 2

Example

```
MOV AX, @data
MOV DS, AX
MOV ES, AX
CLD
MOV SI, OFFSET S_STRING
```

A] LODS S_STRING

B] LODSB

C] LODSW

STOS : Store String.

STOSB : Store String Byte.

STOSW : Store String Word.

- The instruction STOS transfer a byte or a word from the AL for byte and AX for word to destination string pointed by DI in ES.
- On execution of a string instruction, DI is automatically updated to point next element of the source string.
- If DF = 0, the register DI will be incremented by 1 for byte and incremented by 2 for word.
- If DF=1, the register DI will be decremented by 1 for byte and decremented by 2 for word.
- In the instruction STOS, the source must be explicitly declared either with DB or with DW.
- Another way to store a byte or word string is by using implicit instruction STOSB and STOSW.
- The instruction STOSB is used to transfer a byte from AL to destination and the instruction STOSW is used to transfer a word from AX to destination.
- In multiple byte or word loads, the count must be loaded in CX register which functions as a counter.

Operation

If byte movement
ES:[DI] ← AL
For DF = 0 DI ← DI + 1
For DF = 1 DI ← DI – 1
If word movement
ES:[DI] ← AX
For DF = 0 DI ← DI + 2
For DF = 1 DI ← DI – 2

Example

```
MOV AX, @data
MOV DS, AX
MOV ES, AX
CLD
MOV DI, OFFSET D_STRING
A]   STOS D_STRING
B]   STOSB
C]   STOSW
```

CMPS : Compare String.

CMPSB : Compare String Byte.

CMPSW : Compare String Word.

- The instruction CMPS compares a byte or a word in the source string with a byte or word in the destination string.
- The source must be in the data segment and destination must be in extra segment.
- The offset of the source byte or a word must be placed in SI register, which is represented as DS:SI and offset of the destination byte or a word must be in DI register, which is represented as ES:DI.
- On the execution of this instruction, SI and DI register are automatically incremented by one to point next element of source and destination.
- If the direction flag is reset [DF=0], the register SI and DI will be incremented by one for **byte compare** and incremented by two for **word compare**.
- If the direction flag is set [DF=1], the register SI and DI will be decremented by one for **byte compare** and decremented by two for **word compare**.
- The DS:SI and ES:DI register must be loaded prior to the execution of CMPS instruction.
- In the instruction STOS, the source must be explicitly declared either with DB or with DW.
- Another way to compare a byte or word string is by using implicit instruction CMPSB and CMPSW.
- The instruction CMPSB is used to compare a byte in source with destination and the instruction CMPSW is used to compare a word in source destination.
- In multiple byte or word compare, the count must be loaded in CX register which functions as a counter.

Flags modified : AF, CF, OF, PF, SF, ZF

Operations performed by the instruction :

1. If [destination string byte/word > source string byte/word] then CF = 0, ZF = 0, SF = 0
2. If [destination string byte/word < source string byte/word] then CF = 1, ZF = 0, SF = 1
3. If [destination string byte/word = source string byte/word] then CF = 0, ZF = 1, SF = 0

For byte comparison

1. If DF = 0 then SI ← SI + 1, DI ← DI + 1
2. If DF = 1 then SI ← SI – 1, DI ← DI – 1

For word comparison

1. If DF = 0 then SI ← SI + 2, DI ← DI + 2
2. If DF = 1 then SI ← SI – 2, DI ← DI – 2

Examples

```
MOV AX, @data
MOV DS, AX
MOV ES, AX
CLD
MOV SI, OFFSET S_STRING
MOV DI, OFFSET D_STRING
A]   CMPS S_STRING, D_STRING
B]   CMPSB
C]   CMPSW
```

SCAS : Scan String

SCASB : Scan String Byte

SCASW : Scan String Word

- The instruction SCAS scans a string byte or a word with byte in AL and word in AX.
- The destination string is pointed by DI in ES.

- On execution of a string instruction, DI is automatically updated to point next element of the source string.
- If DF = 0, the register DI will be incremented by 1 for byte and incremented by 2 for word.
- If DF = 1, the register DI will be decremented by 1 for byte and decremented by 2 for word.
- In the instruction STOS, the source must be explicitly declared either with DB or with DW.
- Another way to scan a byte or word in a string is by using implicit instruction SCASB and SCASW.
- The instruction SCASB is used to scan a byte in string and the instruction SCASW is used to scan a word in string.
- In multiple byte or word scan, the count must be loaded in CX register which functions as a counter.

Flags modified : AF, CF, OF, PF, SF, ZF

Operations performed by the instruction

1. [If byte in AL or word in AX > destination string byte or word] then CF = 0, ZF = 0, SF = 0
2. [If byte in AL or word in AX < destination string byte or word] then CF = 1, ZF = 0, SF = 1
3. [If byte in AL or word in AX = destination string byte or word] then CF = 0, ZF = 1, SF = 0

For byte scan

1. If DF = 0 then DI ← DI + 1
2. If DF = 1 then DI ← DI – 1

For word scan

1. If DF = 0 then DI ← DI + 2
2. If DF = 1 then DI ← DI – 2

Examples

```
MOV AX, @data
MOV DS, AX
MOV ES, AX
CLD
MOV DI, OFFSET D_STRING
MOV AL, 'V'
A]   SCAS  D_STRING
B]   SCASB
C]   SCASW
```

REP : Repeat

- The instruction is used as a prefix instruction with the string instructions and interpreted as a "repeat while not end of string" [CX not equal to Zero].
- Count for repeat must be loaded in CX register.

Operation

While CX ≠ 0
1. Execute string instruction
2. CX ← CX – 1

Examples

```
MOV AX, @data
MOV DS, AX
MOV ES, AX
CLD
MOV CX, length of string
MOV SI, OFFSET S_STRING
MOV DI, OFFSET D_STRING
REP CMPSB or CMPSW
```

REPE : Repeat while equal.

REPZ : Repeat while zero.

- The instruction is used as a prefix instruction with the string instructions and interpreted as a "repeat while not end of string and string equal" [CX not equal to Zero and ZF = 1].
- Count for repeat must be loaded in CX register.
- Assembler generate same machine code for the instruction REPZ and REPE.

Operation

While CX ≠ 0 and ZF = 1
1. Execute string instruction
2. CX ← CX – 1

Examples

```
        MOV AX, @data
        MOV DS, AX
        MOV ES, AX
        CLD
        MOV CX, length of string
              MOV SI, OFFSET S_STRING
              MOV DI, OFFSET D_STRING
              REPE CMPSB or CMPSW
              JE STR_EQU
NOT_EQU :     MOV AX, 01H
        .
        .
        .
STR_EQU :     MOV AX, 00H
        .
        .
```

REPNE : Repeat while not equal.

REPNZ : Repeat while not zero.

- The instruction is used as a prefix instruction with the string instructions and interpreted as a "repeat while not end of string and string not equal" [CX not equal to Zero and ZF = 0].
- Count for repeat must be loaded in CX register.
- Assembler generates same machine code for the instruction REPNZ and REPNE.

Operation

While CX ≠ 0 and ZF = 0
1. Execute string instruction
2. CX ← CX – 1

Examples

```
MOV AX, @data
MOV DS, AX
MOV ES, AX
```

```
        CLD
        MOV CX, length of string
        MOV SI, OFFSET S_STRING
        MOV DI, OFFSET D_STRING
            REPE CMPSB or CMPSW
            JNE NOT_EQU
STR_EQU : MOV AX,01H
        :
NOT_EQU : MOV AX, 00H
```

Example 3.4.1 S-14, 4 Marks

If AL, BL and CL contain 10H, 10H and 20H respectively. State the effect of following instructions.

Solution :

(a) CMP BL, CL : CF = 1, ZF = 0, SF = 1 as compare instruction perform non-destructive subtract operation which indicate BL < CL.

(b) XCHG AL, CL : After the execution AL contain 20H and CL contain 10H

Example 3.4.2 W-14, 2 Marks

Write assembly language instructions of 8086 microprocessor to

(a) Add 100H to contents of AX register

(b) Rotate the contents of AX towards left by 2 bits

Solution :

(a) Add 100H to contents of AX register:

```
ADD AX, 100H
```

(b) Rotate the contents of AX towards left by 2 bits

```
MOV CL, 2
ROL AX, CL
```

Example 3.4.3 W-16, 4 Marks

Write the appropriate 8086 instructions to perform the following operation.

(i) Multiply AL register contents by 4 using shift instructions.

(ii) Move 2000 H into CS register.

Solution :

(i) Multiplying AL register contents by 4 using shift instructions

```
MOV CL,02H
SHL AL,CL
```

(ii) Moving 2000 H into CS register

```
MOV AX, 2000H
MOV CS, AX
    OR
MOV AX, 2000H
PUSH AX
POP CS
```

Example 3.4.4 S-16, 4 Marks

Write assembly language instructions of 8086 microprocessor to

(1) Divide the content of AX register by 50H

(2) Rotate the content of BX register by 4 bit toward left.

Solution :

(1) Divide the content of AX register by 50H

```
MOV BL, 50H
DIV BL
```

(2) Rotate the content of BX register by 4 bit toward left

```
MOV CL, 04H
ROL BX, CL or RCL BX, CL
```

Example 3.4.5 S-16, 4 Marks

Write an instructions of 8086 to perform following operations.

(i) Shift the content of BX register 3 bit toward left.

(ii) Move 1234H in DS register.

Solution :

(i) To shift the content of BX register 3 bit toward left

```
MOV CL, 03H
SHL BX, CL or SAL BX, CL
```

(ii) To move 1234H in DS register

```
MOV AX, 1234H
MOV DS, AX
```

Example 3.4.6 S-16, 4 Marks

How many times LOOP1 will be executed in the following program. Write the content of AL register after the execution of following program.

```
MOV CL, 00H
MOV AL, 00H
LOOP1: ADD AL, 01H
DEC CL
JNZ LOOP1
```

Solution :

- In above program, initial counter value in CL is 0 and when the instruction DEC CL is executed first time, the value in CL register became FFH.
- Hence the Loop will be executed 256 Times. The Value of CL will be decremented from FFH to 00H.
- The content of AL is '0' and it is incremented by 1 hence the value of AL will go from 00H to FFH but after the last iteration the Value of AL will be 00H.
- Therefore, the value of AL = 00H and CL = 00H

Example 3.4.7 W-16, 4 Marks

Write 8086 assembly language instruction for the following:

(i) Move 5000H to register DX

(ii) Multiply AL by 05H

Solution :

(i) Move 5000H to register DX

```
MOV DX, [5000H]
```

(ii) Multiply AL by 05H :

```
MOV BL, 05H
MUL BL
```

Example 3.4.8 S-17, 4 Marks

Write assembly language instruction of 8086 microprocessor to

(i) Copy 1000H to register BX

(ii) Rotate register BL left four times

Solution :

(i) Copy 1000H to register BX

```
MOV BX, 1000H
```

(ii) Rotate register BL left four times

```
MOV CL,4
ROL BL, CL OR  RCL BL,CL
```

Example 3.4.9 S-17, 4 Marks

What will be the content of register AL after the execution of last instruction ?

```
MOV AL, 02H
MOV BL, 02H
SUB AL, BL
MUL 08H
```

Solution :

```
MOV AL, 02H ;    Loads 02H byte in AL register
MOV BL, 02H ;    Loads 02H byte in BL Register
SUB AL, BL  ;    Subtract BL from AL, AL becomes 00H
MUL 08H     ;    MUL instruction cannot used in
            ;    immediate addressing mode
```

Hence, during the execution of MUL 08H instruction, you will get the error.

Example 3.4.10 S-17, 4 Marks

Write an appropriate 8086 instruction to perform following operation :

(i) Initialize stack of 4200H

(ii) Multiply AL by 05H

Solution :

(i) Initialize stack of 4200H

```
MOV AX, 4200H
MOV SS, AX
```

(ii) Multiply AL by 05H

```
MOV BL,05H
MUL BL
```

Example 3.4.11 W-17, 4 Marks

Write assembly language instruction of 8086 microprocessor to

(i) Multiply 4H by 5H

(ii) Rotate content of AX by 4 bit towards left

Solution :

(i) Multiply 4H by 5H

```
MOV AL,04H
MOV BL,05H
MUL BL
```

(ii) Rotate content of AX by 4 bit towards left

```
MOV CL,4
ROL AX, CL OR  RCL AX,CL
```

Example 3.4.12 W-17, 4 Marks

What will be the content of register BX after execution of instruction ?

```
MOV BX, 2050H
MOV CL, 05H
SHL BX, CL
```

Solution :

- MOV BX, 2050 instruction loads BX register with value 2050H
- MOV CL, 05H instruction load CL register with value 05H
- SHL BX, CL instruction shifts the content of BX register toward left by 5 bits as given below.

	Shift left																
	BX Register = 2050H ←																
Out	D_{15}	D_{14}	D_{13}	D_{12}	D_{11}	D_{10}	D_9	D_8	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	CL=5
	0	0	1	0	0	0	0	0	0	1	0	1	0	0	0	0	
0	0	1	0	0	0	0	0	0	1	0	1	0	0	0	0	0	5
0	1	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	4
1	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	3
0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	2
0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	1
	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	Result
	Finally result in BX = 0A00H after shifting five times toward left																

Example 3.4.13 S-18, 4 Marks

Write assembly language instruction of 8086 microprocessor to.

(i) Add 100 H to the contents of AX register.

(ii) Rotate the contents of AX towards left by 2 bits.

Solution :

(i) Add 100 H to the contents of AX register.

```
MOV AX, 100H
```

(ii) Rotate the contents of AX towards left by 2 bits.

MOV CL,02H

ROL AX, CL OR RCL AX,CL

Example 3.4.14 S-18, 4 Marks

How many times LOOP1 will be executed in following program? What will be the contents of BL after the execution ?

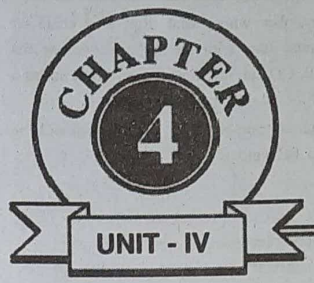
```
        MOV BL, 00H
        MOV CL, 05H
LOOP1 : ADD BL, 02 H
        DEC CL
        JNZ LOOP1.
```

Solution :

- In above program, initial counter value in CL is 05H and when the instruction DEC CL is executed first time, the value in CL register became 04H.
- Hence the LOOP1 will be executed 5 Times. The Value of CL will be decremented from 05H to 00H.
- The content of BL is '0' and after adding 02H five times the content of BL will be 0AH.

Chapter Ends...

❑❑❑



8086 Assembly Language Programming

Syllabus

Model of 8086 Assembly Language Programs. Programming using assembler : Arithmetic operations on Hex and BCD numbers, Sum of series, Smallest and Largest numbers from array, Sorting numbers in Ascending and Descending order, Finding ODD, EVEN positive and negative numbers in the array, Block transfer String operations Length, Reverse, Compare, Concatenation, Copy, Count numbers of '1' and '0' in 16 bit number.

Syllabus Topic : Model of Assembly Language Programs

4.1 Model of Assembly Language Programming

→ **(MSBTE - W-15, W-16, W-17)**

Q. 4.1.1 Write the program structure for writing program in assembly language with suitable comment. (Ref. sec. 4.1) W-15, 4 Marks

Q. 4.1.2 Describe the model of assembly language programming. (Ref. sec. 4.1) W-16, W-17, 4 Marks

Now, we will see the structure of assembly language program of 8086. The general assembly language programs of 8086 are given below.

Model 1 : Using SEGMENT, ASSUME and ENDS directives

```
My_Data   SEGMENT
      :
      :
   Program Data Declaration
   [Data Segment of the program]
      :
      :
My_Data   ENDS
My_Code   SEGMENT
     ASSUME CS:My_Code, DS:My_Data
      MOV AX, My_Data   ;Initialization of the data segment
      MOV DS, AX
      :
   Program Codes [Code Segment of the program]
      :
      :
My_Code ENDS
     END
```

- In above model, My_Data is the name of the data segment used to declare data of the program along with its data type i.e. DB, DW etc.
- My_Code is the name of the code segment used to write code of the program or task to perform.
- The END indicates termination of the program.

Model 2 : Using .DATA and .CODE directives

```
.MODEL SMALL
.STACK 100
.DATA
  :
  :
   Program Data Declaration
   [Data Segment of the program]
  :
  :
.CODE
   MOV AX, @DATA    ;Data Segment initialization
   MOV DS,AX
   :
   :
   Program Codes [Code Segment of the program]
   :
   :
   ENDS
   END
```

- In above model, **.model small** indicate memory model to be used in the program.
- **.STACK** 100 indicate 100 word memory location are reserved for the stack.
- **.DATA** indicates start of the data segment where data declaration of the program is made.
- **.CODE** indicates start of the code segment by actual code of the program.
- The **ENDS** directive indicate end of the data and code segment and **END** indicate termination the program.

4.2 Symbols, Variables and Constants

- Variables are symbols whose value can be dynamically varied during the run time.
- The assembler assigns a memory location to a variable, which is not visible for the user directly and translate the assembly language statements to the machine language statements.
- Variable name should be selected such that the name reflects the meaning of the value it holds.
- For example, a meaning variable name for storing the age of the person is 'age' or 'person_age'.
- So, any variable can be used but meaningful variable names increase the readability of the program and ease of maintenance.

The Assembler symbols consist of the following characters.

Upper case alphabets	A – Z	Lower case alphabets	a – z
Digits	0 – 9	Special Characters_ , @, $, ?	

☞ Rules for variable names

(a) Variable name can have any of the following characters A – Z , a – z, 0 – 9, _(Underscore), @

(b) A variable name must start with a letter (alphabets) or an underscore. So digit can not be used as a first character.

(c) The length of a variable name depends on the assembler, normally, the maximum length is 32 characters.

(d) There is no distinction between the upper and lower case letters. Hence, the variable names are case insensitive in 80 × 86 assembly language.

Examples of valid variable

num1, age, my_data, array, result_lsb, count, average, small, large, next_i, name

total_marks, rate, baud_rate etc.

☞ Numeric Constants

- The numeric constants can be represented as binary or decimal or hexadecimal integer.
- The binary numbers must end with the letter 'B', the decimal numbers must end with the letter 'D' and the hexadecimal numbers must end with the letter 'H'.
- However, the number which does not end with either the letter 'D', 'B' or 'H' is treated as a decimal number.
- The valid binary constant must have only digit 0 or 1, the valid decimal constant must have only digit 0 to 9 and the hexadecimal constant must have both 0 to 9 and A to F.
- The hexadecimal number whose first digit is a character between A to F must start with the digit 0, because the number like BA, A0, CD etc. are treated as symbol, not as a numeric constant.
- Hence the hexadecimal number BA, A0, CD should be written as 0BA, 0A0, 0CD etc.

Examples

Valid Constants

10101010	; Decimal constant
10101010D	; Decimal constant
10101010B	; Binary constant
10101010H	; Hexadecimal constant
8965H	; Hexadecimal constant
0A5B6H	; Hexadecimal constant

Invalid constants

10102101B	; 2 is not binary digit
09A7	; A is not a decimal digit
ABC8	; treated as symbol

Syllabus Topic : Programming using Assembler

4.3 Programming using Assembler

4.3.1 Addition of Two Numbers

➔ **(MSBTE - W-14, S-17)**

Q. 4.3.1 Write an assembly language program to add two 16 bit numbers. (Ref. sec. 4.3.1) **W-14, S-17, 4 Marks**

➢ **Program 1 : Addition of two 8 bit numbers**

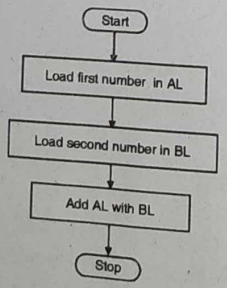
(a) Assume two 8 bit numbers are stored in AL and BL registers of 8086 CPU

AL = 80H and BL = 70H

Algorithm

1. Load first number in AL.
2. Load second number in BL.
3. Add AL with BL.
4. Stop.

Flowchart : Refer Flowchart 1.



Flowchart 1

Program 1(a)

```
.model small
.code
    mov al, 80H    ;Load first number in AL
    mov bl, 70H    ;Load second numbers in BL
    add al, bl     ;Add AL with BL
    ends           ;Stop
    end
```

- In this program, it is assumed that numbers are available in AL and BL so no data segment declaration is required.
- The output of the program can be seen using debugger program **DEBUG.**

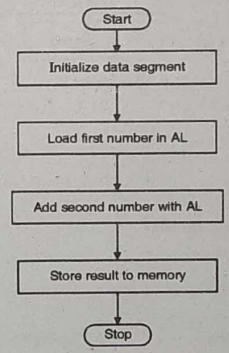
(b) Assume two 8 bit numbers are stored in memory using variable name **num1** and **num2** respectively. Store result in memory location using variable **res.**

num1 = 80H and num2 = 08H

Algorithm

1. Initialize data segment
2. Load first number from memory in AL.
3. Add Second number with first number.
4. Store result in memory location.
5. Stop.

Flowchart : Refer Flowchart 2.



Flowchart 2

Program 1(b)

```
.model small
.data
    num1    db  80H    ;First Number
    num2    db  08H    ;Second Number
    res db  ?          ;Result Variable
.code
    mov ax,@data       ;Initialization of data segment
    mov ds, ax
    mov al, num1       ;load 1st number in AL
    add al, num2       ;add 2nd number with 1st
                       number in AL
    mov res, al        ;Store result from al to memory
                       ; location
    ends
    end
```

- In above program, the memory location for num1, num2 and res will be allocated by the assembler itself, so for that data segment declaration is required.
- So using **.data** directive, data segment can be declared with variables required in the program.
- In program these variable or constants name can be used to refer data associated with it.

➢ **Program 2 : Addition of two 16 bit numbers**

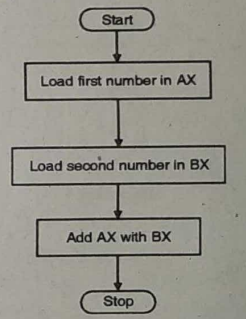
(a) Assume two 16 bit numbers are stored in AX and BX registers of 8086 CPU

AX = 8960H and BX = 7C60H

Algorithm

1. Load first number in AX.
2. Load second number in BX.
3. Add AX with BX.
4. Stop.

Flowchart : Refer Flowchart 3.



Flowchart 3

Program 2(a)

```
.model small
.code
mov ax, 8960H          ;Load first number in AX
mov bx, 7C60H          ;Load second numbers in BX
add ax, bx             ;Add AX with BX
ends                   ;Stop
end
```

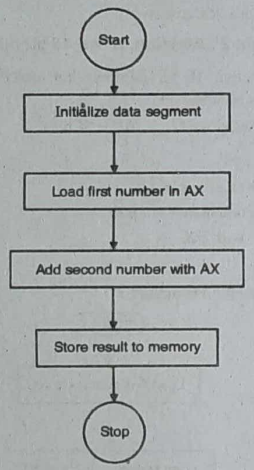
- In this program, it is assumed that numbers are available in AX and BX so no data segment declaration is required.
- The output of the program can be seen using debugger program **DEBUG.**

(b) Assume two **16** bit numbers are stored in memory using variable name **num1** and **num2** respectively. Store result in memory location using variable **res**.
num1 = 8A64H and num2 = 5F98H.

Algorithm

1. Initialize data segment.
2. Load first number from memory in register.
3. Add second number with first number.
4. Store result in memory location.
5. Stop.

Flowchart : Refer Flowchart 4.



Flowchart 4

Program 2(b)

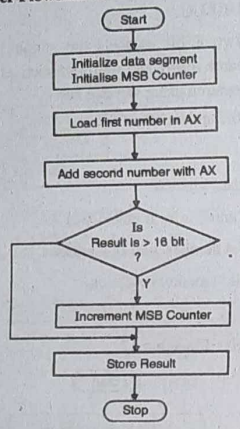
```
.model small
.data
    num1    dw    8A64H    ;First Number
    num2    dw    5F98H    ;Second Number
    resdw?                 ;Result Variable
.code
    mov ax,@data   ;Initialization of data segment
    mov ds,ax
    mov ax, num1   ;load 1st number in AL
    add ax, num2   ;add 2nd number with 1st
                   number in AX
    mov res, ax    ;Store result from al to
                   memory location
    ends
    end
```

➢ **Program 3 : Addition of two 16 bit numbers where result may be more than 16 bit.**

Algorithm

1. Initialize data segment.
2. Load first number in register.
3. Add second number with first number.
4. Check result > 16 bit if yes, then go to step 5 else step 6.
5. Increment MSB counter.
6. Store result.
7. Stop.

Flowchart : Refer Flowchart 5.



Flowchart 5

Program 3

```
.model small
.data
    num1    dw   0ffffh
    num2    dw   0ffffh
    res_lsb dw   0
    res_msb dw   0
.code
    mov ax,@data    ;Initialize data segment
    mov ds,ax
    mov ax,num1     ;load num1 to AX
    add ax,num2     ;Add num1 and num2
    jnc exit        ;if result > 16 bit
    inc res_msb     ;increment carry counter
exit:
    mov res_lsb,ax  ;store result
    ends
    end
```

– In this program result of the addition may be greater than 16 bit, so two result variable are declared in the data segment i.e. res_lsb and res_msb whose data type is DW.

4.3.2 Subtraction of Two Numbers

➔ **(MSBTE - S-16, S-17, W-17)**

Q. 4.3.2 Write an assembly language program to subtract two 16 bit numbers. (Ref. sec. 4.3.2 - Program 5(a)) **S-16, 4 Marks**

Q. 4.3.3 Write an ALP to subtract two 8 bit numbers. (Ref. sec. 4.3.2) **S-17, W-17, 4 Marks**

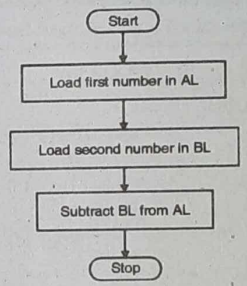
➢ **Program 4 : Subtraction of two 8 bit numbers.**

(a) Assume two 8 bit numbers are stored in AL and BL registers of 8086 CPU
AL = 80H and BL = 70H

Algorithm

1. Load first number in register.
2. Load second number in another register.
3. Subtract second number from first number.
4. Stop.

Flowchart : Refer Flowchart 6



Flowchart 6

Program 4(a)

```
.model small
.code
    mov al, 80H      ;Load first number in AL
    mov bl, 70H      ;Load second numbers in BL
    sub al, bl       ;Subtract BL from AL
    ends             ;Stop
    end
```

– In this program, it is assumed that numbers are available in AL and BL so no data segment declaration is required.

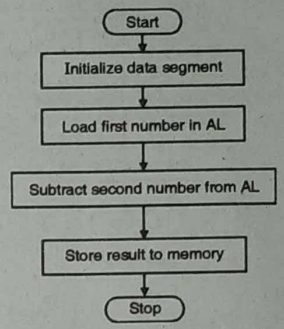
– The output of the program can be seen using debugger program **DEBUG.**

(b) Assume two 8 bit numbers are stored in memory using variable name **num1** and **num2** respectively. Store result in memory location using variable **res**.
num1 = 80H and num2 = 08H

Algorithm

1. Initialize data segment
2. Load first number from memory in register.
3. Subtract 2nd number from 1st number.
4. Store result in memory location.
5. Stop.

Flowchart : Refer Flowchart 7



Flowchart 7

Program 4(b)

```
.model small
.data
    num1        db     80H     ;First Number
    num2        db     08H     ;Second Number
    res     db  ?              ;Result Variable
.code
    mov ax,@data               ;Initialization of data segment
    mov ds,ax
    mov al, num1               ;load 1st number in AL
    sub al, num2               ;subtract 2nd number from 1st
                               ;number
    mov res, al                ;Store result from al to memory
                                location
    ends
    end
```

– In this program, the memory location for num1, num2 and res will be allocated by the assembler itself, so for that data segment declaration is required.

– So using **.data** directive, data segment can be declared with variables required in the program.

– In program, these variable or constants name can be used to refer data associated with it.

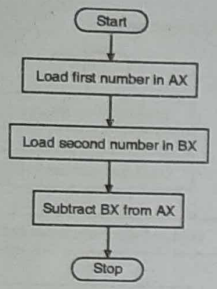
➢ **Program 5 : Subtraction of two 16 bit numbers.**

(a) Assume two 16 bit numbers are stored in AX and BX registers of 8086 CPU
AX = 8960H and BX = 7C60H

Algorithm

1. Load first number in register.
2. Load second number another register.
3. Subtract second number from first number.
4. Stop.

Flowchart : Refer Flowchart 8.



Flowchart 8

Program 5(a)

```
.model small
.code
mov ax, 8960H     ;Load first number in AX
mov bx, 7C60H     ;Load second numbers in   BX
sub ax, bx        ;Subtract BX from AX
ends              ;Stop
end
```

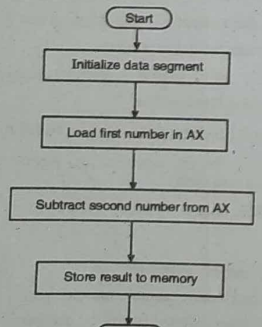
- In this program, it is assumed that numbers are available in AX and BX so no data segment declaration is required.
- The output of the program can be seen using debugger program **DEBUG.**

(b) Assume two 16 bit numbers are stored in memory using variable name **num1** and **num2** respectively. Store result in memory location using variable **res**.
num1 = 8A64H and num2 = 5F98H.

Algorithm

1. Initialize data segment.
2. Load first number from memory in register.
3. Subtract 2nd number from 1st number.
4. Store result in memory location.
5. Stop.

Flowchart : Refer Flowchart 9.



Flowchart 9

Program 5(b)

```
.model small
.data
    num1      dw    8A64H  ;First Number
    num2      dw    5F98H  ;Second Number
    res       dw    ?      ;Result Variable
.code
    mov ax,@data   ;Initialization of data segment
    mov ds,ax
    mov ax, num1   ;load 1st number in AL
    sub ax, num2   ;subtract 2nd number from 1st number
    mov res, ax    ;Store result from al to memory location
    ends
    end
```

Syllabus Topic : Sum of Series

4.3.3 Sum Numbers in the Array [SUM of SERIES]

➔ **(MSBTE - W-14, S-17, S-18)**

Q. 4.3.4 Write an ALP to find sum of first 10 integers. (Ref. sec. 4.3.3 - Program 6(c)) S-14, W-16, 4 Marks

Q. 4.3.5 Write an assembly language program to add the series of 5 numbers. (Ref. sec. 4.3.3) W-14, S-18, 4 Marks

Q. 4.3.6 Write an ALP to add BCD numbers in an array of 10 numbers. Assume suitable array with BCD numbers. Store the result at the end of array. (Ref. sec. 4.3.3 - Program 6(c)) S-15, 4 marks

Q. 4.3.7 Write an ALP to find sum of series OBH, 05H, 07H, OAH, 01H. (Ref. sec. 4.3.3) S-17, 4 Marks

- Here we will see the addition of the numbers in the series or array of n numbers which are stored in the memory.
- So, byte or word counter which indicate length of array in the term of byte or word, is required to read numbers from the array.
- The result of addition may be greater than either 8 bit or 16 bit depending on numbers stored in the array.
- The memory pointer must be initialized to read byte or word from the array of n numbers.
- In following program, the numbers in the array are 8 bit numbers, so the sum of these numbers may be greater than 8 bit.
- For that we check CF, if CF is set, then result is greater than 8 bit and can not store in 8 bit register or variable.

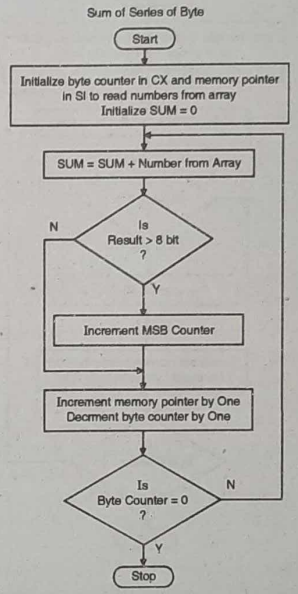
- So another register or variable should be taken to store higher byte of result as demonstrate in following program.

➢ **Program 6(a) : Addition of five 8 bit numbers in series result may be greater than 16 bit.**

Algorithm

1. Initialize data segment.
2. Initialize byte counter and memory pointer to read number from array.
3. Initialize sum variable with 0.
4. Sum = sum + number from array.
5. If sum > 8 bit then go to step 6 else step 7.
6. Increment MSB result counter.
7. Increment memory pointer.
8. Decrement byte counter.
9. If byte counter = 0 then step 10 else step 4.
10. Stop.

Flowchart : Refer Flowchart 10.



Flowchart 10

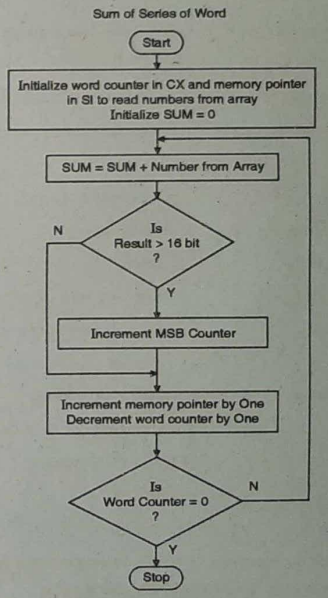
Program 6(a)

```
.model small
.data
    array     db  0ffh,0ffh,0ffh,0ffh,0ffh
    sum_lsb   db  0
    sum_msb   db  0
.code
    mov ax,@data          ;Initialize data segment
    mov ds,ax
    mov cx,5              ;Initialize byte counter
    mov si,offset array   ;Initialize memory pointer
up:
    mov al,[si]           ;Read byte from memory
    add sum_lsb,al        ;Add with sum
    jnc next              ;If Sum>8 bit
    inc sum_msb           ;Increment msb counter
next:
    inc si                ;Increment memory pointer
    loop up               ;decrement byte counter
                          ;if byte counter = 0 then exit
                          ;else read next number
    ends
    end
```

- In this program, array consists of five bytes, so byte counter is initialized in CX register with 5.
- Then, memory pointer is initialized using SI register.
- The instruction **mov si, offset array** loads starting offset address of the array in SI register.

➢ **Program 6(b) : Add five 16 bit numbers in series result may be greater than 32 bit.**

Flowchart : Refer Flowchart 11.



Flowchart 11

Algorithm

1. Initialize data segment.
2. Initialize word counter and memory pointer to read number from array.
3. Initialize sum variable with 0.
4. Sum = sum + number from array.
5. If sum > 16 bit then go to step 6 else step 7.
6. Increment MSB result counter.
7. Increment memory pointer.
8. Decrement word counter.
9. If word counter = 0 then step 10 else step 4.
10. Stop.

Program 6(b)

```
.model small
.data
        array       dw  0ffffh,0ffffh,0ffffh,0ffffh,0ffffh
        sum_lsw     dw  0
        sum_msb     db  0
.code
    mov ax,@data        ;Initialize data segment
    mov ds,ax
    mov cx,5            ;Initialize word counter
    mov si,offset array ;Initialize memory pointer
up:
    mov ax,[si]         ;Read word from memory
    add sum_lsw,ax      ;Add with sum
    jnc next            ;If Sum >16 bit
    inc sum_msb         ;Increment msb counter next:
    inc si              ;Increment memory pointer
    inc si
    loop up             ;decrement word counter
                        ;if word counter = 0 then exit
                        ;else read next number
    ends
    end
```

- After the execution of the above program, the result will be available in sum_lsw and sum_msb variable.

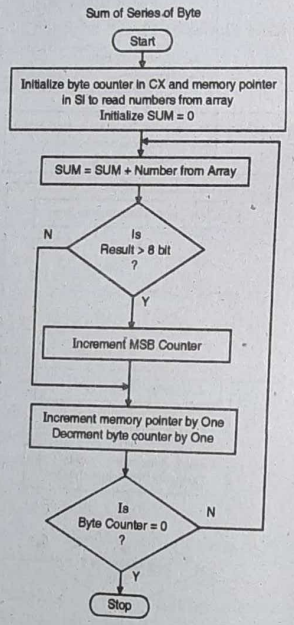
➢ **Program 6(c) : Addition of 10 BCD Numbers in Series**

➔ **(MSBTE - S-14, S-15, W-16)**

Algorithm

1. Initialize data segment.
2. Initialize byte counter and memory pointer to read number from array.
3. Initialize sum variable with 0.
4. Sum = sum + number from array.
5. Adjust result to BCD
6. If sum > 8 bit then go to step 6 else step 7.
7. Increment MSB result counter.
8. Increment memory pointer.
9. Decrement byte counter.
10. If byte counter = 0 then step 10 else step 4.
11. Stop.

Flowchart



Program 6(c)

```
.model small
.data
        array       db  1,2,3,4,5,6,7,8,9,10
        sum_lsb     db  0
        sum_msb     db  0
.code
    mov ax,@data        ;Initialize data segment
    mov ds,ax
    mov cx,10           ;Initialize byte counter
    mov si,offset array ;Initialize memory pointer
up:
    mov al,[si]         ;Read byte from memory
    add sum_lsb,al      ;Add with sum
    daa
    jnc next            ;If Sum>8 bit
    inc sum_msb         ;Increment msb counter
next:
    inc si              ;Increment memory pointer
    loop up             ;decrement byte counter
                        ;if byte counter = 0 then exit
                        ;else read next number
    ends
```


- In this program, array consists of ten BCD numbers, so byte counter is initialized in CX register with 10.
- Then, memory pointer is initialized using SI register.
- The instruction **mov si, offset array** loads starting offset address of the array in SI register.

4.3.4 Multiplication of Unsigned and Signed Numbers

➔ **(MSBTE - S-18)**

Q. 4.3.8 Write an ALP for 8086 to multiply two 16 bit numbers. (Ref. sec. 4.3.4 - Program 7(b)) **W-15, S-16, 4 Marks**

Q. 4.3.9 Write an assembly language program to multiply two 8 bit number. (Ref. sec. 4.3.4) **S-18, 4 Marks**

☞ **Unsigned and signed data**

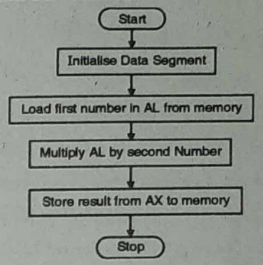
- For unsigned data, all bits are intended to be data bits.
- Hence maximum number is 255 in decimal and FFH in hexadecimal for 8 bit number in binary and 65535 in decimal and FFFFH in hexadecimal.
- For signed data, the leftmost bit is a sign bit and remaining bits are data bits.
- Hence, the range of value that can represent in 8 bit signed number is –128 to 127 and – 32768 to 32767 for 16 bit signed number.
- For multiplication, the MUL instruction handles unsigned data and the IMUL instruction handles signed data.

➢ **Program 7(a) : Program to multiply two 8 bit unsigned numbers result is max. 16 bit.**

Algorithm

1. Initialize data segment.
2. Load first number.
3. Multiply first number with second number.
4. Store result.
5. Stop.

Flowchart : Refer Flowchart 12.



Flowchart 12

Program 7(a)

```
.model small
.data
        num1    db  0ffh
        num2    db  0ffh
        result  dw  0
.code
    mov ax,@data        ;Initialize data segment
    mov ds,ax
    mov al,num1         ;Multiply num1 by num2
    mul num2
    mov result,ax       ;Store result
    ends
    end
```

- In above program, num1 and num2 are two 8 bit unsigned numbers and result will be 16 bit as large as possible.
- We will get result in AX register which is 16 bit.

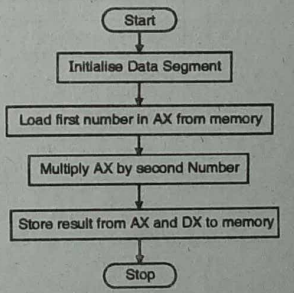
➢ **Program 7(b) : Program to multiply two 16 bit unsigned numbers result is max 32 bit.**

➔ **(MSBTE - W-15, S-16, S-18)**

Algorithm

1. Initialize data segment.
2. Load first number.
3. Multiply first number with second number.
4. Store result.
5. Stop.

Flowchart : Refer Flowchart 13.



Flowchart 13

Program 7(b)

```
.model small
.data
   num1    dw   0ffffh
   num2    dw   0ffffh
   res_lsb dw   0
   res_msb dw   0
.code
   mov ax,@data        ;Initialize data segment
   mov ds,ax
   mov ax,num1         ;Multiply num1 by num2
   mul num2
   mov res_lsb,ax      ;Store LSB of result
   mov res_msb,dx      ;Store MSB of result
   ends
   end
```

- In above program, num1 and num2 are two 16 bit unsigned numbers and result will be 32 bit as large as possible.
- We will get result in **DX:AX** registers which are two 16 bit registers.

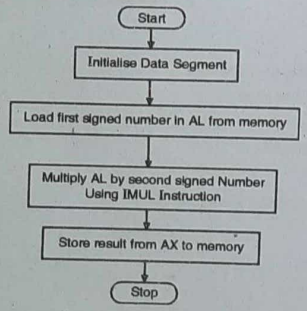
➢ Program 7(c) : Program to multiply two 8 bit signed numbers result is max. 16 bit.

Case 1 : Both numbers are negative [Signed numbers]

Algorithm

1. Initialize data segment.
2. Load first number.
3. Multiply first number with second number.
4. Store result.
5. Stop.

Flowchart : Refer Flowchart 14.



Flowchart 14

Program 7(c)(1)

```
.model small
.data
   num1    db   –5h
   num2    db   –4h
   result  dw   0
.code
   mov ax,@data        ;Initialize data segment
   mov ds,ax
   mov al,num1         ;Multiply num1 by num2
   imul num2
   mov result,ax       ;Store result
   ends
   end
```

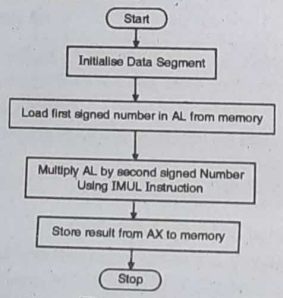
- In this program, num1 and num2 are two 8 bit signed numbers and result will be 16 bit as large as possible.
- We will get result in AX register which is 16 bit register.
- The result of above program is **0014H** in AX which is positive.

Case 2 : One number is positive [unsigned] and another number is negative [signed].

Algorithm

1. Initialize data segment.
2. Load first number.
3. Multiply first number with second number.
4. Store result.
5. Stop.

Flowchart : Refer Flowchart 15.



Flowchart 15

Program 7(c)(2)

```
.model small
.data
   num1    db   –5h
   num2    db   4h
   result  dw   0
.code
   mov ax,@data        ;Initialize data segment
   mov ds,ax
   mov al,num1         ;Multiply num1 by num2
   imul num2
   mov result,ax       ;Store result
   ends
   end
```


- In above program, num1 is 8 bit signed number and num2 is 8 bit unsigned number and result will be 16 bit as large as possible.
- We will get result in AX register which is 16 bit register.
- The result of above program is **FFECH** in AX which is 2nd complement of 14H [– 20 in Decimal] as 2nd complement method is used to represent negative hexadecimal number.

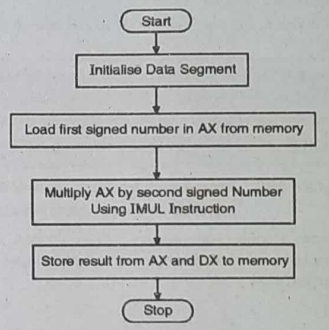
➢ Program 7(d) : Program to multiply two 16 bit signed numbers result is max. 32 bit.

Case 1 : Both numbers are negative [signed numbers]

Algorithm

1. Initialize data segment.
2. Load first number.
3. Multiply first number with second number.
4. Store result.
5. Stop.

Flowchart : Refer Flowchart 16.



Flowchart 16

Program 7(d)(1)

```
.model small
.data
   num1    dw  –12H    ;[–18 in decimal]
   num2    dw  –10H    ;[–16 in decimal]
   res_lsw dw  0
   res_msw dw  0
.code
   mov ax,@data        ;Initialize data segment
   mov ds,ax
   mov ax,num1         ;Multiply num1 by num2
   imul num2
   mov res_lsb,ax      ;Store LSB of result
   mov res_msb,dx      ;Store MSB of result
   ends
   end
```

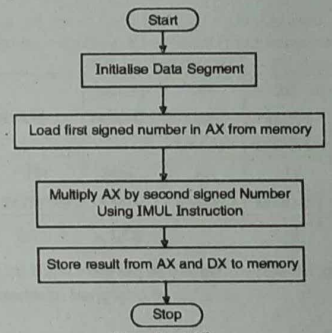
- In above program, num1 and num2 are two 16 bit unsigned numbers and result will be 32 bit as large as possible.
- We will get result in **DX:AX** registers which are two 16 bit registers.
- The result of the above program is **0000:0120H** in **DX:AX** which is positive.

Case 2 : One number is positive [unsigned] and another number is negative [signed].

Algorithm

1. Initialize data segment
2. Load first number.
3. Multiply first number with second number.
4. Store result.
5. Stop.

Flowchart : Refer Flowchart 17.



Flowchart 17

Program 7(d)(2)

```
.model small
.data
   num1    dw  –12H    ;[–18 in decimal]
   num2    dw  10H     ;[16 in decimal]
   res_lsb dw  0
   res_msb dw  0
.code
   mov ax,@data        ;Initialize data segment
   mov ds,ax
   mov ax,num1         ;Multiply num1 by num2
   imul num2
   mov res_lsb,ax      ;Store LSB of result
   mov res_msb,dx      ;Store MSB of result
   ends
   end
```

- In above program, num1 and num2 are two 16 bit unsigned numbers and result will be 32 bit as large as possible.
- We will get result in **DX:AX** registers which are two 16 bit registers.

- The result of the above program is **FFFF:FEE0H** in **DX:AX** which is 2nd complement of 120H [–288 in decimal].

4.3.5 Division of Unsigned and Signed Numbers

➔ **(MSBTE - S-14, W-16, W-17, S-18)**

Q. 4.3.10 Write an ALP to divide two 16 bit unsigned numbers. (Ref. sec. 4.3.5) S-14, S-18, 4 Marks

Q. 4.3.11 Write an assembly language program to perform word by byte division of two unsigned number. (Ref. sec. 4.3.5) W-16, 4 Marks

Q. 4.3.12 Write ALP to divide two 16 bit numbers. (Ref. sec. 4.3.5) W-17, 4 Marks

- For division, the DIV instruction handles unsigned data and IDIV instruction handles signed data.
- The basic divide operations are word by byte, double word by word.
- The dividend must be in AX for word by byte divide operation and DX:AX for double word by word operation as shown in Fig. 4.3.1(a) and Fig. 4.3.1(b).

Before division	DX Higher dividend	AX Lower dividend
After division	DX Reminder	AX Quotient

(a)

Before division	AX dividend	
After division	AH Remainder	AL Quotient

(b)

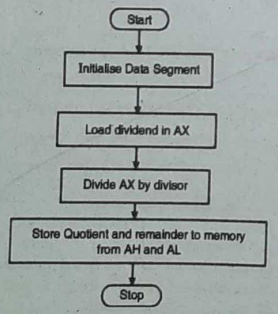
Fig. 4.3.1

➢ **Program 8(a) : Program to perform word by byte division of unsigned numbers.**

Algorithm

1. Initialize data segment.
2. Load first number.
3. Divide first number by second number.
4. Store quotient and remainder.
5. Stop.

Flowchart : Refer Flowchart 18.



Flowchart 18

Program 8(a)

```
.model small
.data
    dividend dw  0123h
    divisor  db  12h
    quo      db  0
    rem      db  0
.code
     mov ax,@data        ;Initialize data segment
    mov ds,ax
      mov ax, dividend   ;Divide word bit by byte
     div divisor
     mov quo,al          ;Store Quotient
     mov rem,ah          ;Store Remainder
    ends
    end
```

- In above program, dividend is word i.e. 16 bit number and divisor is byte i.e. 8 bit number.
- After division, the remainder is 03 in AH and quotient is 10 in AL.

Note : Remember that quotient should not greater than 8 bit as AH register is 8 bit which stores quotient otherwise you will find divide overflow error on the screen while debugging or executing the program.

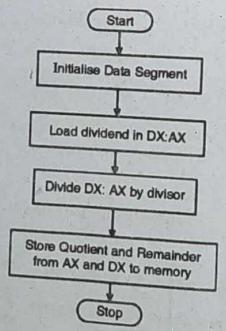
➢ **Program 8(b) : Program to perform double word by word division of unsigned numbers.**

Case 1 : Now, Double word number can be defined using two separate variable name for lower and higher word as given below.

Algorithm

1. Initialize data segment.
2. Load first number.
3. Divide first number by second number.
4. Store quotient and remainder.
5. Stop.

Flowchart : Refer Flowchart 19.



Flowchart 19

Program 8(b)(1)

```
.model small
.data
    div_lsw   dw  0ffffh ;Lower word of dividen
    div_msw   dw  0001h ;Higher word of dividend
    divisor   dw  0fh
    quo       dw  0
    rem       dw  0
.code
    mov ax,@data;Initialize data segment
    mov ds,ax
    mov ax,div_lsw      ;load LSW of dividend
    mov dx,div_msw      ;load MSW of dividend
    div divisor         ;Divide double word by word
    mov quo,ax          ;Store Quotient
    mov rem,dx          ;Store Remainder
    ends
    end
```

- After the execution of above program, the quotient is 2222H in AX and the remainder 0001H in DX.

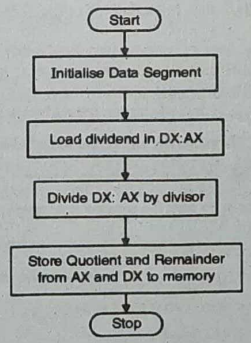
Note : Remember that quotient should not greater than 16 bit as AX register is 16 bit which stores quotient otherwise you will find divide overflow error on the screen while debugging the program.

Case 2 : Now, Double word number can be defined using single variable name having data type of DD and can be loaded in DX and AX using directive WORD and PTR as given as follows.

Algorithm

1. Initialize data segment.
2. Load first number.
3. Divide first number with second number.
4. Store quotient and remainder.
5. Stop.

Flowchart : Refer Flowchart 20.



Flowchart 20

Program 8(b)(2)

```
.model small
.data
    dividend dd  0001FFFFH     ;Dividend as a single
                               variable
    divisor  dw  0FH
    quo      dw  0
    rem      dw  0
.code
    mov ax, @data              ;Initialize data
                               segment
    mov ds, ax
    mov ax, word ptr dividend  ;load LSW of Dividend
    mov dx, word ptr dividend+2 ;Load MSW of
                               Dividend
    div divisor                ;Divide Double Word
                               by word
    mov quo, ax                ;Store quotient
    mov rem, dx                ;Store remainder
    ends
    end
```

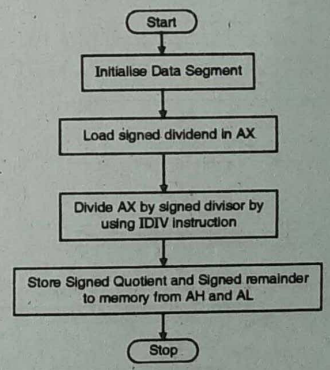
- In this program, mov ax, word ptr dividend instruction loads lower word of the dividend and mov dx, word ptr dividend+2 loads higher word of the dividend.
- The result will be same as that of first case program.

➢ **Program 8(c) : Program to perform word by byte division of signed numbers.**

Algorithm

1. Initialize data segment.
2. Load first number.
3. Divide first number with second number.
4. Store quotient and remainder.
5. Stop.

Flowchart : Refer Flowchart 21.



Flowchart 21

Program 8(c)

```
.model small
.data
    dividend dw  –123h
    divisor  db  12h
    quo      db  0
    rem      db  0
.code
    mov ax,@data        ;Initialize data segment
    mov ds,ax
    mov ax, dividend    ;Divide word bit by byte
    idiv divisor
    mov quo,al          ;Store Quotient
    mov rem,ah          ;Store Remainder
    ends
    end
```

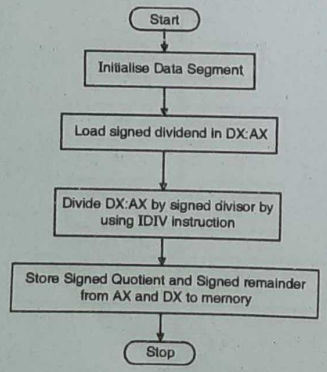
- In above program, dividend is negative word i.e. 16 bit negative number and divisor is byte i.e. 8 bit number.
- After division, the remainder is FD i.e. 2nd complement of 03 [– 03H] in AH and quotient is F0H i.e. 2nd complement of 10 [–10H] in AL.
- So, the microprocessor gives negative result in 2nd complement representation of the corresponding numbers.
- The 2nd complement of 03 is FDH and 10 is F0H as negative numbers are represented in binary as a 2nd complement of that number.

➢ **Program 8(d) : Program to perform double word by word division of signed numbers.**

Algorithm

1. Initialize data segment.
2. Load first number.
3. Divide first number with second number.
4. Store quotient and remainder.
5. Stop.

Flowchart : Refer Flowchart 22.



Flowchart 22

Program 8(d)

```
.model small
.data
    dividend dd  – 12345H;Dividend as a single variable
    divisor  dw  12H
    quo      dw  0
    rem      dw  0
.code
    mov ax, @data           ;Initialize data segment
    mov ds, ax
    mov ax, word ptr dividend     ;load LSW of Dividend
    mov dx, word ptr dividend+2   ;Load MSW of Dividend
    idiv divisor            ;Divide Double Word by  word
    mov quo, ax             ;Store quotient
    mov rem, dx             ;Store remainder
    ends
    end
```

- After the execution of above program, the quotient is EFD2H in AX which 2nd complement of 102EH i.e. –102E and the remainder should be 0009H in DX but we will find FFF7H which is 2nd complement of 0009H i.e. – 0009H.

➢ **Program 8(e) : Program to perform word by word [16 bit by 16 bit] division of unsigned numbers.**

- As we know that dividend must be 32 bit number if divisor is 16 bit number.
- But still, we can perform word by word division of unsigned as well as signed numbers.
- So, the dividend which is 16 bit number must be converted to 32 bit number.
- There are two ways as given below :
  1. By loading DX with 0000H or FFFF depending on D_{15} bit of AX and AX with 16 bit dividend.
  2. By using CWD instruction.
- Normally, use second method, because CWD instruction extend sign bit of 16 bit number in AX to DX and maintain the sign of the number.
- In first method you must know D_{15} bit of AX is either 0 or 1, accordingly DX must be loaded with either 0000H or FFFFH.

For example

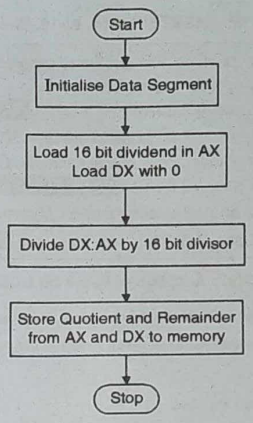
If AX = 0123H then after the execution of CWD instruction DX will be loaded with 0000H as D_{15} bit of AX is 0 which is sign bit for 16 bit signed number indication number is positive.

If AX = 8123H then after the execution of CWD instruction DX will be loaded with FFFFH as D_{15} bit of AX is 1 which is sign bit for 16 bit signed number indication number is negative.

Algorithm

1. Initialize data segment.
2. Load first number.
3. Divide first number with second number.
4. Store Quotient and remainder.
5. Stop.

Flowchart : Refer Flowchart 23.



Flowchart 23

Program 8(e)

```
.model small
.data
    dividend    dw    1234H
    divisor     dw    0012H
    quo         dw    0
    rem         dw    0
.code
mov ax, @data       ; Initialize data segment
mov ds, ax
mov ax, dividend    ; load dividend in AX
cwd                 ;Convert word to double word with sign bit
                    ; DX = 0000H and AX = 1234H
div divisor         ; Divide double word by word
mov quo, ax         ; store quotient
mov rem, dx         ; store remainder
ends
end
```

- In above program, after the execution of the CWD instruction, the content of DX will be 0000H as D_{15} bit of AX i.e. dividend is 0 and AX will be 1234H.
- The result will be 0102H in AX i.e. quotient and 0010H in AX i.e. remainder.
- For signed division, dividend or divisor should be taken as a negative number in a data segment as given below :

```
Dividend    dw  –7123H
    OR/AND
divisor     dw  –1245H
```

- Then, execute above program with IDIV instruction instead of DIV instruction, we will get negative result.

➢ **Program 8(f) : Program to perform byte by byte [8 bit by 8 bit] division of unsigned numbers.**

- As we know that dividend must be 16 bit number if divisor is 8 bit number.
- But still, we can perform byte by byte division of unsigned as well as signed numbers.
- So, the dividend which is 8 bit number must be converted to 16 bit number.
- There are two ways as given below :
  1. By loading AH with 00H or FF depending on D_7 bit of AL and AL with 8 bit dividend.
  2. By using CBW instruction.
- Normally, use second method, because CBW instruction extend sign bit of 8 bit number in AL to AH and maintain the sign of the number.
- In first method you must know D_7 bit of AL is either 0 or 1, accordingly AH must be loaded with either 00H or FFH.

For example

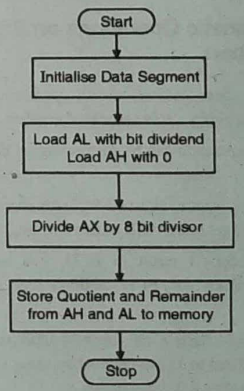
If AL = 23H then after the execution of CBW instruction AH will be loaded with 00H as D_7 bit of AL is 0 which is sign bit for 8 bit signed number indication number is positive.

If AL = 81H then after the execution of CBW instruction AH will be loaded with FFH as D_7 bit of AL is 1 which is sign bit for 8 bit signed number indication number is negative.

Algorithm

1. Initialize data segment
2. Load first number.
3. Divide first number with second number.
4. Store Quotient and remainder.
5. Stop.

Flowchart : Refer Flowchart 24.



Flowchart 24

Program 8(f)

```
.model small
.data
    dividend db  23H
    divisor  db  12H
    quo      db  0
    rem      db  0
.code
    mov ax, @data       ; Initialize data segment
    mov ds, ax
    mov al, dividend    ; load dividend in AL
    cbw                 ; Convert byte to word
                        with sign bit
                        ; AH = 00H and AL = 23 H
    div divisor         ;Divide word by byte
    mov quo, al         ; store quotient
    mov rem, ah         ; store remainder
    ends
    end
```

- In above program, after the execution of the CBW instruction, the content of AH will be 00H as D_7 bit of AL i.e. dividend is 0 and AX will be 0023H.
- The result will be 01H in AL i.e. quotient and 11H in AH i.e. remainder.
- For signed division, dividend or divisor should be taken as a negative number in a data segment as given below.

```
dividend    db    – 81H
OR/AND
divisor     db    – 80H
```

- Then, execute above program with IDIV instruction instead of DIV instruction, we will get negative result.

4.3.6 Arithmetic Operations on BCD Numbers

- We know that, microprocessor perform all arithmetic operation on binary i.e. hexadecimal numbers.
- Arithmetic operation can be performing on BCD number but not directly.
- Suppose, you want to add two BCD numbers i.e. 84 and 28, then you will get result ACH which is wrong result.
- So, to get correct result in BCD, you have to use the instruction DAA for BCD addition and DAS for BCD subtraction.
- Now in above example, the result of addition is ACH which can be converted to 112 in decimal by using DAA instruction immediately after ADD instruction.
- The execution of DAA and DAS instruction has been discussed in chapter 3.

4.3.6(A) Addition of BCD Numbers

➔ **(MSBTE - W-14, W-15, W-16, S-17,S-18)**

Q. 4.3.13 Write an assembly language program to add two BCD numbers using 8086. (Ref. sec. 4.3.6.(A)) **W-14, 4 Marks**

Q. 4.3.14 Write an ALP for 8086 to perform BCD addition of two numbers. [Assume suitable data]. (Ref. sec. 4.3.6.(A)) **W-15, W-16, S-17, 4 Marks**

Q. 4.3.15 Write an ALP to add 16 bit BCD number. (Ref. sec. 4.3.6(A)) **S-18, 4 Marks**

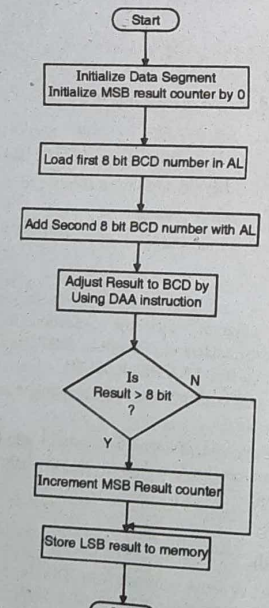
➢ **Program 9(a) : Addition of two 8 bit BCD number**

Assume, two numbers are declared in a data segment as ***num1=84H*** and ***num2=28H*** as given in following program.

Algorithm

1. Initialize data segment.
2. Initialize MSB counter with 0.
3. Load first BCD number in AL.
4. Add second BCD number with AL.
5. Adjust result to BCD.
6. If result > 8 bit then goto step 7 else 8.
7. Increment MSB counter by 1.
8. Store result.
9. Stop.

Flowchart : Refer Flowchart 25.



Flowchart 25

Program 9(a)

```
.model small
.data
    num1   db  84H    ;First 8 bit BCD Number
    num2   db  28H    ; Second 8 bit BCD Number
    res_lsb db  0     ; Variable to store LSB of result
    res_msb db  0     ;Variable to store MSB of result
.code
    mov ax, @data     ; Initialize data segment
    mov ds, ax
    mov al, num1      ; Load first BCD number in AL
    add al, num2      ; Add second BCD number with first
    daa               ; Adjust result to correct BCD
    jnc dn            ; if result > 8 bit if no go to dn
    inc res_msb       ; else increment MSB result counter
dn : mov res_lsb, al
    ends
    end
```

- In above program, after the execution of ADD instruction, the result will be AC in AL.
- But, when DAA will gets executed, the result will be adjusted to correct BCD result as 112 where LSB of result i.e. **12** will be stored at variable **res_lsb** and **01** at **res_msb**.
- This result can be checked by debugging the above program.

➢ **Program 9(b) : Addition of two 16 bit BCD numbers**

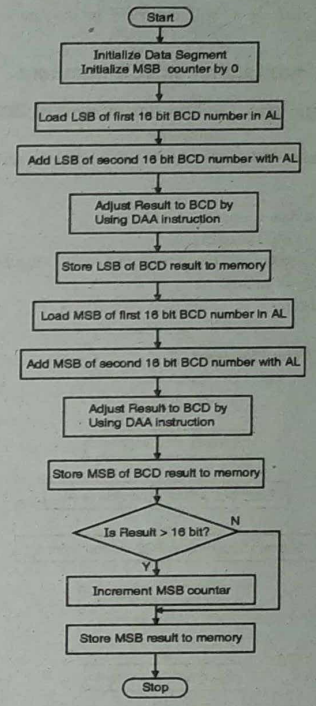
- As we know DAA operates only on AL only which is 8 bit register.
- So, in addition of two 16 bit BCD numbers, ADD and DAA instructions must be used two time for LSB and MSB addition of BCD number as given in following program.

➢ **Program to ADD two 16 bit BCD numbers**

Algorithm

1. Initialize data segment and MSB byte counter with 0.
2. Load lower byte of first 16 bit BCD number.
3. Add lower byte of first BCD number with lower byte of second BCD number.
4. Adjust result to BCD.
5. Store result of lower byte's addition.
6. Load higher byte of first 16 bit BCD number.
7. Add higher byte of first BCD number with higher byte of second BCD number.
8. Adjust result to BCD.
9. Store result of higher byte's addition.
10. If result > 16 bit then go to step 11 else 12.
11. Increment MSB result counter.
12. Stop.

Flowchart : Refer Flowchart 26.



Flowchart 26

Program 9(b)

```
.model small
.data
    num1   dw  9999h
    num2   dw  9999h
    res_lsw dw  0
    res_msb db  0
.code
    mov ax,@data          ;Initialise data segment
    mov ds,ax
    mov al,byte ptr num1          ;Add LSB first
    add al,byte ptr num2          ;convert result to BCD
    daa
    mov byte ptr res_lsw,al       ;Store LSB of result
    mov al,byte ptr num1+1
    adc al,byte ptr num2+1        ;Add MSB next
    daa                           ;Convert result to BCD
    mov byte ptr res_lsw+1,al     ;Store result
    jnc exit                      ;Check result>16bit
    inc res_msb           ;if yes, increament res_msb by1
exit :  ends
    end
```

- After the execution of the this program, the result will be 19998 where you will find **9998** in **res_lsw** and **01** in **res_msb.**

4.3.6(B) Subtraction of BCD numbers

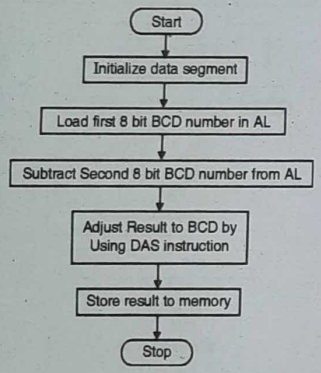
➢ **Program 10(a) : Subtraction of two 8 bit BCD numbers**

Assume to BCD numbers are num1 = 95H and num2 = 19H.

Algorithm

1. Initialize data segment.
2. Load first BCD number.
3. Subtract second BCD number from first BCD number.
4. Adjust result to BCD.
5. Store result.
6. Stop.

Flowchart : Refer Flowchart 27.



Flowchart 27

Program 10(a)

```
Program
.model small
.data
    num1    db   85H    ;First BCD number
    num2    db   57H    ;Second BCD number
    resdb   0           ;Result variable
    .code
    mov ax, @data       ;Initialize data segment
    mov ds, ax
    mov al, num1        ;load first number in AL
    sub al, num2        ;Subtract second no. from first
    das                 ;Adjust result to Correct BCD
    mov res, al         ;Store result
    ends
    end
```

- In this program, after the execution of SUB instruction, the result will be **2E** in AL.
- But, when DAS will gets executed, the result will be adjusted to correct BCD result as **28** in AL.

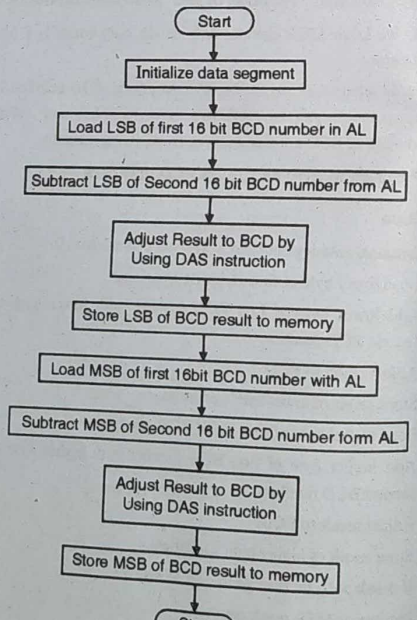
➢ **Program 10(b) : Subtraction of two 16 bit BCD numbers**

- As we know DAS operates only on AL only which is 8 bit register.
- So, in subtraction of two 16 bit BCD numbers, SUB and DAS instructions must be used two time for LSB and MSB subtraction of BCD number as given in following program.

Algorithm

1. Initialize data segment.
2. Load lower byte first 16 bit BCD number.
3. Subtract lower byte of second BCD number from lower byte of first BCD number.
4. Adjust result to BCD.
5. Store result of lower byte's subtraction.
6. Load higher byte of first 16 bit BCD number.
7. Subtract higher byte of second BCD number from higher byte of first BCD number.
8. Adjust result to BCD.
9. Store result of higher byte's subtraction.
10. Stop.

Flowchart : Refer Flowchart 28.



Flowchart 28

Program 10(b)

```
.model small
.data
    num1   dw   9000h
    num2   dw   0999h
    result dw   0
.code
    mov ax,@data;Initialise data segment
    mov ds,ax
    mov al,byte ptr num1      ;Sub LSB first
    sub al,byte ptr num2      ;convert result to BCD
    das
    mov byte ptr result,al    ;Store result
    mov al,byte ptr num1+1
    sbb al,byte ptr num2+1    ;Sub MSB next
    das                       ;Convert result to BCD
    mov byte ptr result+1,al  ;Store result
    ends
    end
```

After the execution of the above program, the correct BCD result will be **8001** in **result** variable.

4.3.6(C) Multiplication of BCD Numbers

Q. 4.3.16 Write an assembly language program to multiply two 8 bit BCD nos. State result in memory. (Ref. sec. 4.3.6(C))

Q. 4.3.17 Write 8086 program to multiply two 8 bit BCD numbers, draw flowchart give explanation and comments. (Ref. sec. 4.3.6(C))

- The multiplication of BCD numbers can not be performed directly because there is no instruction available to adjust result to BCD after multiplication.
- So, the successive addition method can be used to perform BCD multiplication where we can use ADD or ADC and DAA instructions.
- For example, suppose we want to multiply 4 by 5, then we add 4 five times or 5 four times as given below.

$4 \times 5 = 4 + 4 + 4 + 4 + 4$ OR $4 \times 5 = 5 + 5 + 5 + 5$.

- This can implement by taking either 4 or 5 as addition counter in CL for byte or CX for word multiplication.
- If 4 is taken as a addition counter, then 5 can be added 4 times to get correct result of multiplication as given following programs.

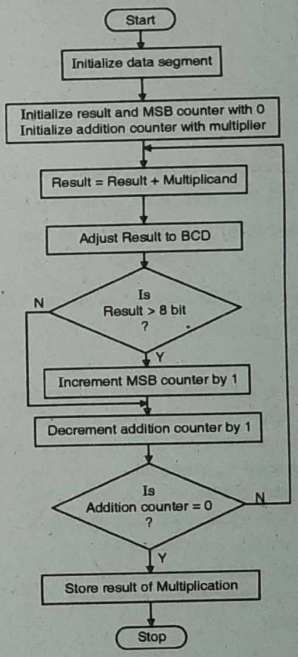
➢ **Program 11(a) : Multiplication of two 8 bit BCD numbers**

Suppose, two BCD number are 09H and 99H, stored in the variables num1 and num2.

Algorithm

1. Initialize data segment and MSB result counter with 0.
2. Load multiplier in Addition counter register.
3. Initialize result with 0.
4. Result = Result + Multiplicand.
5. Adjust result to BCD.
6. If result > 8 bit then go to step 6 else step 7.
7. Increment MSB result counter.
8. Decrement addition counter by one.
9. If addition counter ≠ 0 then go to step 4.
10. Store result.
11. Stop.

Flowchart : Refer Flowchart 29.



Flowchart 29

Program 11(a)

```
.model small
.data
    num1    db   99H
    num2    db   09h
    res_lsb db   0
    res_msb db   0
.code
    mov ax, @data  ;Initialize data segment
    mov ds, ax
    mov cl, num2   ; multiplicand as a addition counter
    mov al, 0      ; initialize AL with 0
up : add al, num1  ; Add num2 to AL num1 times
    daa            ; Adjust result to BCD
    jnc dn         ; Check result is > 8 bit if yes
dn: inc res_msb    ; increment res_msb
    dec cl         ; decrement addition counter
    jnz up         ; if addition counter not zero then
                   go to up
    mov res_lsb, al ;else store LSB of result
    ends
    end
```

- After the execution of above program the result will be 0891 where 91 will be stored in res_lsb and 08 in res_msb variable.

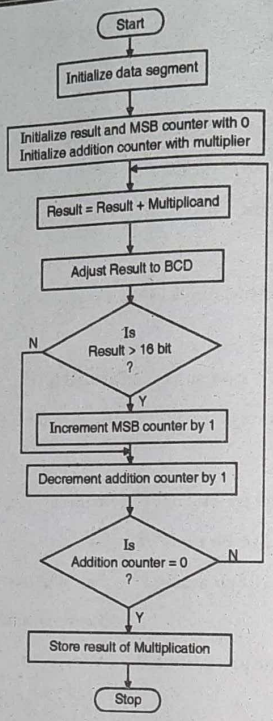
➢ Program 11(b) : Multiplication of two 16 bit BCD numbers

Suppose two BCD numbers are 9H and 9999H stored in variable num1 and num2.

Algorithm

1. Initialize data segment and MSB counter with 0.
2. Load multiplier in Addition counter register.
3. Initialize result with 0.
4. Result = result + multiplicand.
5. Adjust result to BCD.
6. If result > 16 bit then go to step 6 else step 7.
7. Increment MSW result counter.
8. Decrement addition counter by one.
9. If addition counter ≠ 0 then go to step 4.
10. Store result.
11. Stop.

Flowchart : Refer Flowchart 30.



Flowchart 30

Program 11(b)

```
.model small
.data
    num1     dw   9h          ; Multiplier
    num2     dw   9999h       ; Multiplicand
    res_lsw  dw   0
    res_msb dw    0
.code
    mov ax,@data   ; Initialize data Segment
    mov ds,ax
    mov cx,num1    ;Load multiplier as a Counter
    mov ax,0
up :
    add al,byte ptr num2; Multiply two BCD numbers
    daa
    mov byte ptr res_lsw,al  ; Successive Addition method
    mov al,ah
    adc al,byte ptr num2+1
    daa
    mov byte ptr res_lsw+1,al
    mov ax,res_lsw          ; Check result>16 bit
    jnc next
    inc res_msb             ; If yes increment res_msb
next :
    loop up                 ; IF addition counter not 0 go
                            ; to Up
    ends
    end
```

- After the execution of the above program, result will be **89991** where **9991** will be stored in **res_lsw** and **08** in **res_msb**.

4.3.6(D) Division of BCD Numbers

- Same as that of multiplication of BCD numbers, there is no direct instructions are available in instruction set of 8086 for BCD division.
- Hence, **successive subtraction method** can be used to perform BCD division operation for that SUB or SBB and DAS instruction are required.
- In this method, we subtract divisor from dividend until divisor is equal or greater than dividend.
- So, one counter is required to count successive subtraction which gives you quotient of the division operation and the end of the successive subtraction, we will get remainder.
- For example, suppose we want to divide 11 by 2 then it can be perform as given below.

Step 1: Initialize Quotient Counter Q with 0

Step 2 : R = 11 – 02 = 09 Increment Q = Q +1 = 1 Compare Divisor with R,
R > Divisor perform subtraction

Step 3 : R = 09 – 02 = 07 Increment Q = Q +1 = 2 Compare Divisor with R,
R > Divisor perform subtraction

Step 4 : R = 07 – 02 = 05 Increment Q = Q +1 = 3 Compare Divisor with R,
R > Divisor perform subtraction

Step 5 : R = 05 – 02 = 03 Increment Q = Q +1 = 4 Compare Divisor with R,
R > Divisor perform subtraction

Step 6 : R = 03 – 02 = 01 Increment Q = Q +1 = 5 Compare Divisor with R,
R < Divisor stop subtraction

- The value of the Q is 5 which is quotient and the value of R = 1 which remainder of the division using successive subtraction method.
- By writing code using SUB/SBB and DAS instruction, we can perform BCD division as given in following program.

➢ Program 12(a) : Division of two 8 bit BCD numbers

Algorithm

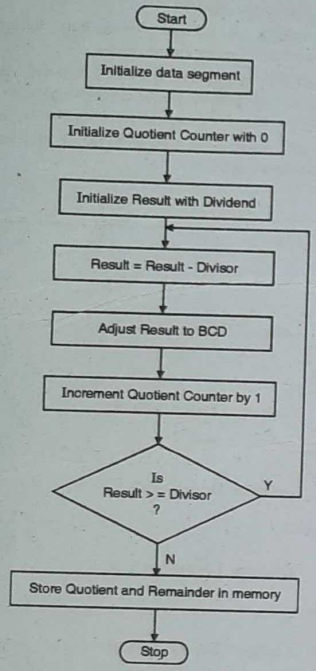
1. Initialize data segment.
2. Initialize quotient counter with 0.
3. Initialize result variable with dividend.
4. Result = Result – Divisor.
5. Adjust result to BCD.
6. Increment quotient counter by 1.
7. If result of subtraction ≥ divisor then go to step 4.
8. Store quotient and remainder available in result.
9. Stop.

Flowchart : Refer Flowchart 31.

Program 12(a)

```
.model small
.data
    divisor    db   02H
    dividend   db   11H
    quo        db   0
    rem        db   0
.code
    mov ax,@data; Initialize data segment
    mov ds,ax
    mov al,dividend    ; division using successive
next: sub al,divisor   ; Subtraction method
    das
    inc quo
    cmp al,divisor
    jnc next
    mov rem,al         ; Store remainder
    ends
    end
```

- After the execution of the above program, the quotient will be **05** in quo variable and remainder is **01** in rem variable.



Flowchart 31

➢ Program 12(b) : Division of two 16 bit BCD numbers

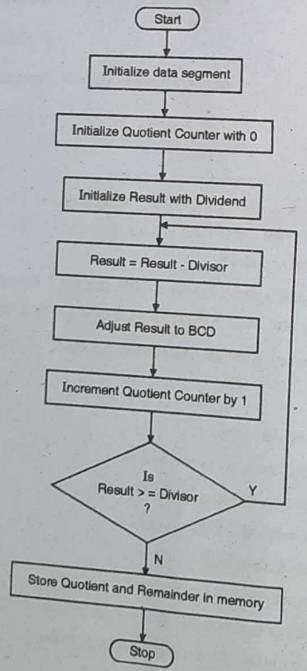
Algorithm

1. Initialize data segment.
2. Initialize quotient counter with 0.
3. Initialize result variable with dividend.
4. Result = Result – Divisor.
5. Adjust result to BCD.
6. Increment quotient counter by 1.
7. If result of subtraction ≥ divisor then go to step 4.
8. Store quotient and remainder available in result.
9. Stop.

Flowchart : Refer Flowchart 32

Program 12(b)

```
.model small
.data
    num1  dw   0009h          ;DIVISOR
    num2  dw   0099h          ;DIVIDEND
    quo   db   0
    rem   dw   0
.code
    mov ax,@data;Initialise data Segment
    mov ds,ax
    mov bh,0
    mov ax,num2
up:
    sub al,byte ptr num1  ;Divide two BCD numbers
    das
    mov byte ptr rem,al       ;Successive Subtraction
                              Methods
    mov al,ah
    sbb al,byte ptr num1+1
    das
    mov byte ptr rem+1,al
    mov al,bl                 ;Convert quotient to BCD
    add al,1
    daa
    mov bl,al
    mov quo,al
    mov ax,rem
    cmp ax,num1    ;Compare result with divisor
    jge up
    ends
    end
```



Flowchart 32

- After the execution of the above program, the quotient will be 0BH in quo variable which converted to 11 in decimal using ADD and DAA instruction.
- Remainder is 00 in **rem** variable.

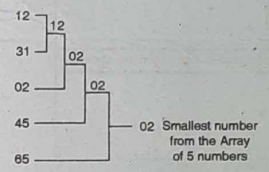
Syllabus Topic : Smallest Number from the Array

4.3.7 Smallest Number from the Array

➔ **(MSBTE - W-14)**

Q. 4.3.18 Write an assembly language program to find smallest number from array of 5 elements. (Ref. sec. 4.3.7) **W-14, 4 Marks**

- Array is the set of N numbers i.e. byte or word.
- So, memory pointer and counter is required to read or write numbers from or to memory location in the array.



- To find smallest number from the array, the numbers in the array must be compared with each other as given as follows.
- Array may consist of 8 bit numbers i.e. byte or 16 bit numbers i.e. word, so memory pointer is required to retrieve numbers from the array.
- As we know, there are 5 bytes in the array but CPU does not know.
- Hence one counter called as byte or word counter which indicates how many numbers are their in the array, must be taken in the program to read and compare only desired numbers from the array.
- Following program demonstrate how to use memory pointer and counter to read numbers from the array.

➢ Program 13(a) : Smallest number from the array of five 8 bit numbers

Algorithm

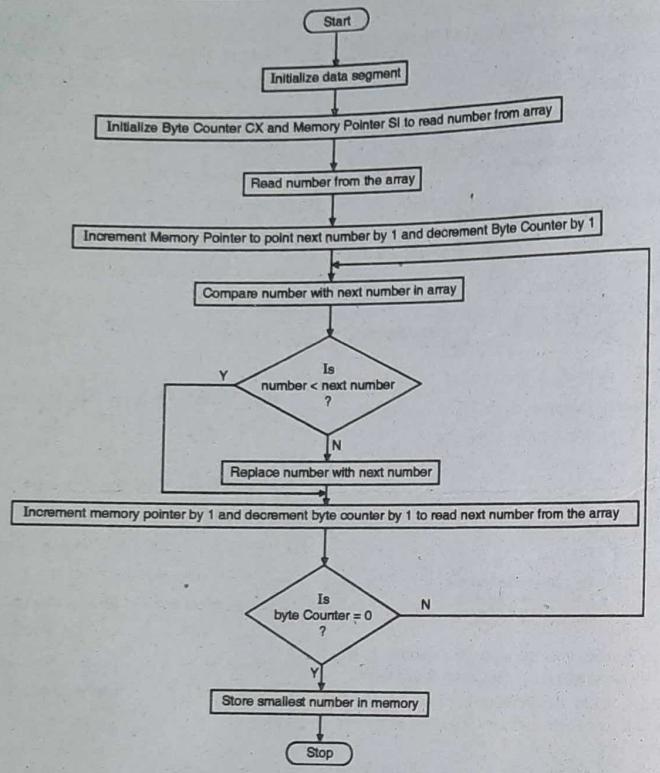
1. Initialize data segment.
2. Initialize byte counter and memory pointer to read numbers from array.
3. Read number from the array.
4. Increment memory pointer to read next number.
5. Decrement byte counter.
6. Compare two numbers.
7. If number < next number then go to step 9.
8. Replace number with next number which is smallest.
9. Increment memory pointer to read next number in the array.
10. Decrement byte counter by 1.
11. If byte counter ≠ 0 then go to step 6.
12. Store result.
13. Stop.

Flowchart : Refer Flowchart 33.

Program 13(a)

```
.model small
.data
    array   db    12h,31h,02h,45h,65h
    small   db    0
.code
    mov ax,@data;Initialize data segment
    mov ds,ax
    mov cx,5      ;Initialize byte counter to read
                  numbers from  array
    mov si,offset array   ;Initialize memory pointer
                          ;to read number
    mov al,[si]           ;read number from the array
    dec cx                ;decrement byte counter by 1
up: inc si                ;increment memory pointer to
                          ;point next number in array
    cmp al,[si]           ;compare numbers to find
                          ;smallest ;number
    jc next               ;if first number < second go to up
    mov al,[si]           ;compare it with next number
next:                     ;decrement byte counter
    loop up               ;if it is NOT ZERO, compare
                        with next number in array
    mov small,al          ;Store smallest number from AL
    ends                  ;memory variable small
    end
```

- In above program, array contains five 8 bit numbers, so CX is loaded with 5 which will act as a byte counter and SI register is used as memory pointer to read number from array.



Flowchart 33

➢ **Program 13(b) : Smallest number from the array of five 16 bit numbers.**

Algorithm

1. Initialize data segment.
2. Initialize word counter and memory pointer to read numbers from array.
3. Read number from the array.
4. Increment memory pointer to read next number.
5. Decrement word counter.
6. Compare two numbers.
7. If first number < second number then go to step 8.
8. Replace first number with second which is smallest.
9. Increment memory pointer to read next number in the array.
10. Decrement word counter by 1.
11. If word counter ≠ 0 then go to step 6.
12. Store result.
13. Stop.

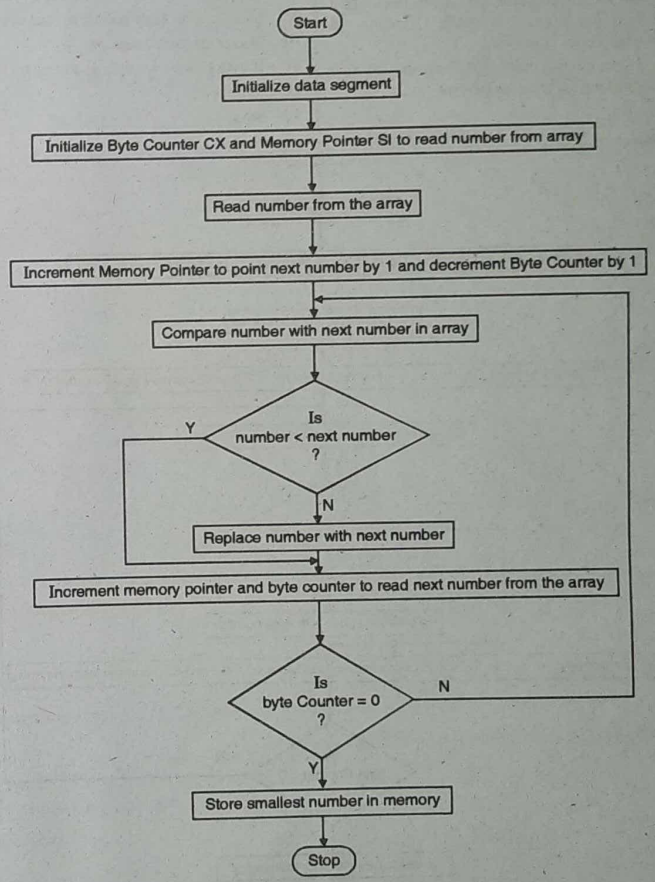
Program 13(b)

```
.model small
.data
    array   dw   12h,31h,02h,45h,65h
    small   dw   0
.code
    mov ax,@data          ;Initialize data segment
    mov ds,ax
    mov cx,5              ;Initialize word counter to
                           read numbers from array
    mov si,offset array   ;Initialize memory
                           pointer to read number
    mov ax,[si]           ;read number from the array
    dec cx                ;decrement word counter by 1
up: inc si                ;increment memory pointer to
                           point next
    inc si                ;number in array
    cmp ax,[si]           ;compare numbers to find
                           smallest number
    jc next               ;if it is smallest then compare
                           it with
    mov ax,[si]           ;next number
next:                     ;decrement word counter
    loop up               ;if it is NOT ZERO, compare
                           with next number in array
    mov small, ax         ;Store smallest number from AX to
    ends                  ;memory variable small
    end
```


– After the execution of above program **a** and **b**, the smallest number are 02H and 0002H respectively.

– The smallest number will be stored in **small** variable.

Flowchart : Refer Flowchart 34.



Flowchart 34

4.3.8 Largest Number from the Array

➔ **(MSBTE - S-15, W-15, W-17)**

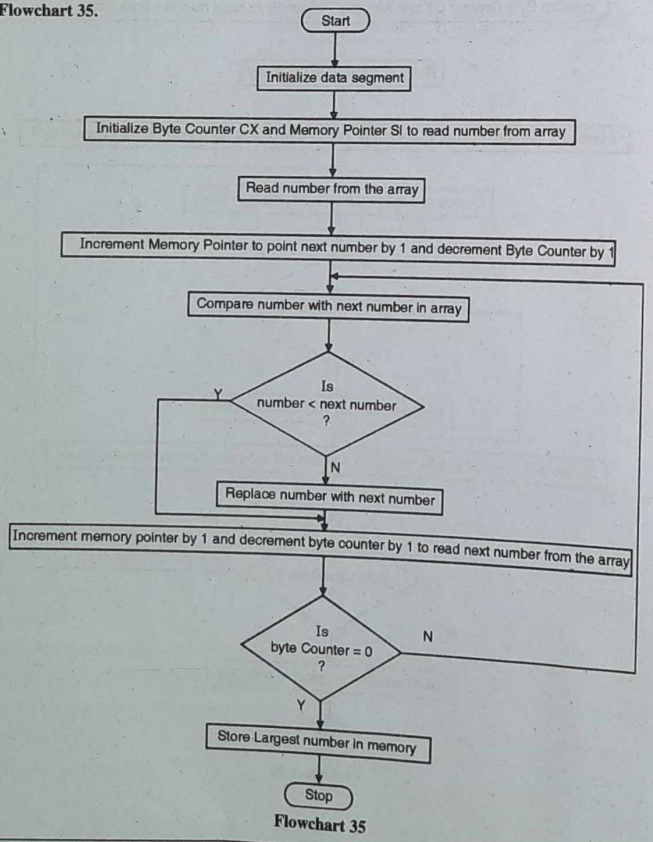
Q. 4.3.19 Write algorithm and assembly language programming for 8086 to find largest number among block of data. Assume size = 15. (Ref. sec. 4.3.8) **S-15, 8 Marks**

Q. 4.3.20 Write an ALP for 8086 to find the largest number in an array. [Assume array size of 10]. (Ref. sec. 4.3.8) **W-15, W-17, 4 Marks**

Q. 4.3.21 Write an assembly language program to find largest number from array of 10 numbers. (Ref.sec. 4.3.8 - Program 14(a)) **S-16, 4 Marks**

- The above program for smallest numbers can be use to find largest number from the array of N numbers, only small change in the above program is required.
- Simply replace **JC** instruction with **JNC**, which will find largest number in the array as given as follows.

Flowchart : Refer Flowchart 35.



Flowchart 35

➤ Program 14(a) : Largest number from the array of five 8 bit numbers

➔ **(MSBTE - S-16)**

Algorithm

1. Initialize data segment.
2. Initialize byte counter and memory pointer to read numbers from array.
3. Read number from the array.
4. Increment memory pointer to read next number.
5. Decrement byte counter.
6. Compare two numbers.
7. If first number > second number then go to step 8.
8. Replace first number with second which is largest.
9. Increment memory pointer to read next number in the array.
10. Decrement byte counter by 1.
11. If byte counter ≠ 0 then go to step 6.
12. Store result.
13. Stop.

Program 14(a)

```
.model small
.data
    array   db  12h,31h,02h,45h,65h
    large   db  0
.code
    mov ax,@data        ;Initialize data segment
    mov ds,ax
    mov cx,5        ;Initialize byte counter to read
                        numbers from array
    mov si,offset array ;Initialize memory pointer to
                        read number
    mov al,[si]         ;read number from the array
    dec cx              ;decrement byte counter by 1
up: inc si              ;increment memory pointer
                    ;to point next number in array
    cmp al,[si]         ;compare numbers to find
                        ;largest number
    jnc next            ;if it is largest then compare
                        ;it with
    mov al,[si]         ;next number
next:                   ;decrement byte counter
    loop up             ;if it is NOT ZERO, compare with
                        next number in array
    mov large,al        ;Store largest number from AL to
    ends                ;memory variable large
    end
```

➤ Program 14(b) : Largest number from the array of five 16 bit numbers.

Algorithm

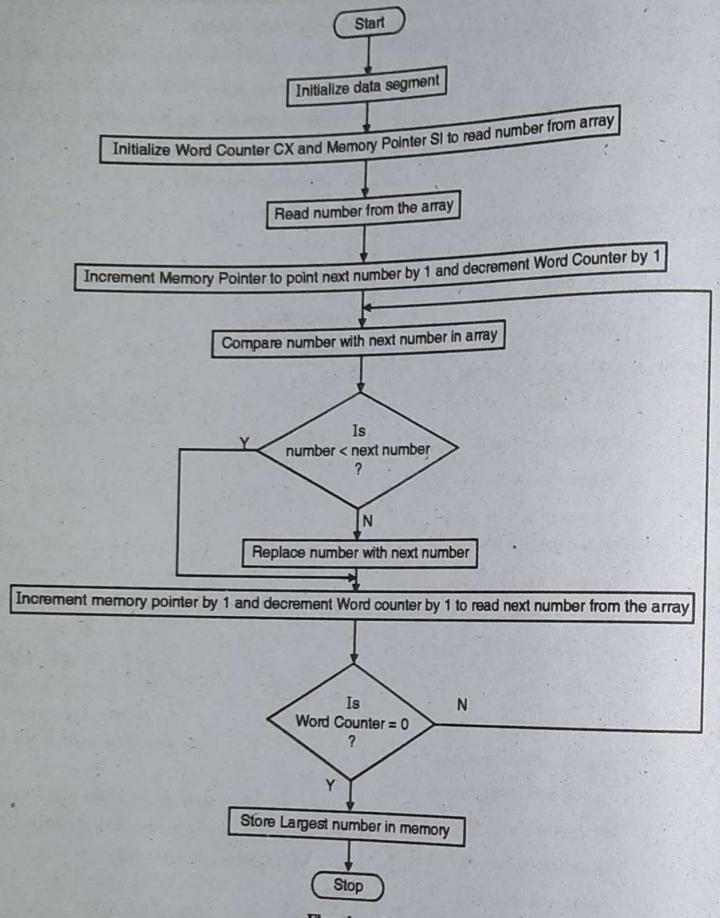
1. Initialize data segment.
2. Initialize word counter and memory pointer to read numbers from array.
3. Read number from the array.
4. Increment memory pointer to read next number.
5. Decrement word counter.
6. Compare two numbers.
7. If first number > second number then go to step 8.
8. Replace first number with second which is largest.
9. Increment memory pointer to read next number in the array.
10. Decrement word counter by 1.
11. If word counter ≠ 0 then go to step 6.
12. Store result.
13. Stop.

Flowchart : Refer Flowchart 36.

Program 14(b)

```
.model small
.data
    array   dw  12h,31h,02h,45h,65h
    large   dw  0
.code
    mov ax,@data        ;Initialize data segment
    mov ds,ax
    mov cx,5            ;Initialize word counter to read
                        ;numbers from array
    mov si,offset array ;Initialize memory pointer to
                        ; read number
    mov ax,[si]         ;read number from the array
    dec cx              ;decrement word counter by 1
up: inc si              ;increment memory pointer
                    ;to point next number in array
    inc si
    cmp ax,[si]     ;compare numbers to find largest number
    jnc next        ;if it is largest then compare it with
    mov ax,[si]     ;next number
next:               ;decrement word counter
    loop up         ;if it is NOT ZERO, compare with next
                    ; number in array
    mov large,ax    ;Store largest number from AX to
    ends            ;memory variable large
    end
```

- After the execution of above program **a** and **b,** the largest number are 65H and 0065H respectively.
- The smallest number will be stored in **large** variable.



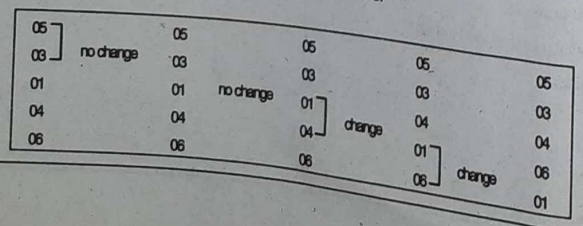
Flowchart 36

4.3.9 Arrange Numbers in the Array In Descending Order

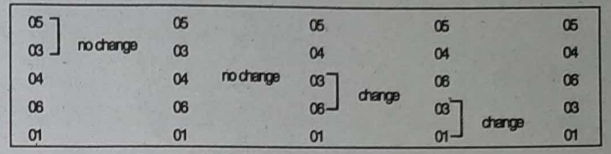
Q. 4.3.22 Write an assembly language program to sort 10 numbers in array in descending order. Draw the flowchart for it. (Ref. sec. 4.3.9 - Program 15) → **(MSBTE - S-16)**

S-16, 8 Marks

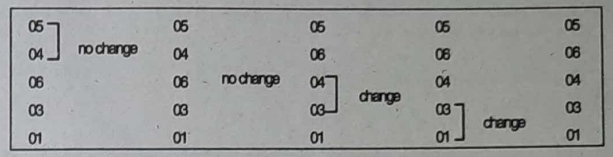
- To arrange numbers in descending order, check two numbers.
- If num1 < num2, then interchange these two numbers.
- Restart same process for remaining numbers in the array as shown as follows.
- Pass 1



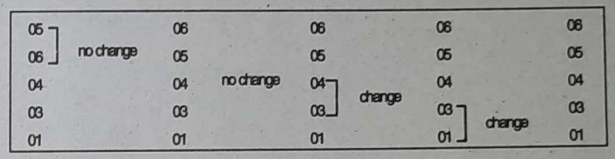
- Pass 2



- Pass 3



- Pass 4

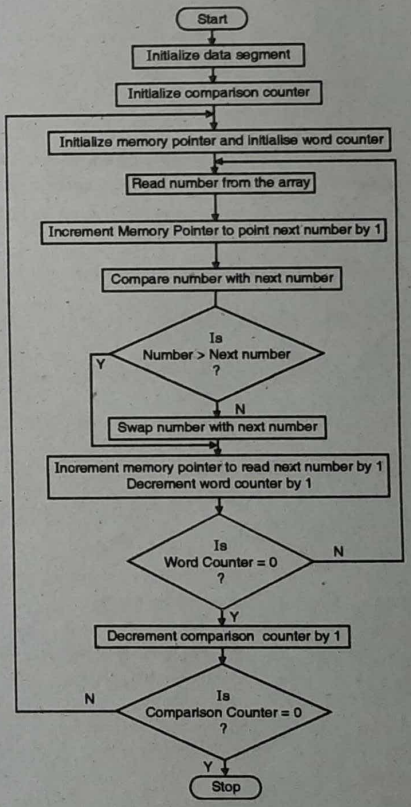


- We have to repeat the process until we get arranged data in descending order and for that two counters are required.
- Where $counter_1$ i.e. **byte or word counter** is needed to compare numbers to find largest among them and $counter_2$ i.e. **pass counter** is needed to repeat this comparison process.

Algorithm

1. Initialize data segment.
2. Initialize comparison or pass counter.
3. Initialize memory pointer to read number from array.
4. Initialize word counter.
5. Read numbers from the array.
6. Compare two numbers.
7. If number ≥ to next number then go to step 9.
8. Interchange or swap numbers.
9. Increment memory pointer to read next number from array.
10. Decrement word counter by one.
11. If word counter ≠ 0 then go to step 5.
12. Decrement comparison counter by one.
13. If comparison counter ≠ 0 then go to step 3.
14. Stop.

Flowchart : Refer Flowchart 37.



Flowchart 37

➢ **Program 15**

```
.model small
.data
    array  dw    12h,11h,21h,9h,19h
.code
    mov ax,@data          ;Initialize data segment
    mov ds,ax
    mov bx,5              ;Initialize pass counter
                          ;i.e. comparison Counter
up1 :
    mov si,offset array   ;Initialize memory pointer
    mov cx,4              ;Initialize word counter
up :
    mov ax,[si]
    cmp ax,[si+2]         ;Compare two numbers
    jnc dn                ;if number > next number
                          ;then go to dn
    xchg ax,[si+2]        ;interchange numbers
    xchg ax,[si]
dn:  add si,2             ;increment memory pointer
    loop up               ;decrement word counter
                          ;if ≠ 0 then up
    dec bx                ;decrement pass counter
                          ;if ≠ 0 then up1
    jnz up1
    ends
    end
```

- After the execution of the above program, all five **16 bit numbers** i.e. word will gets arranged in descending order.
- Same program can be used to arrange **8 bit numbers** i.e. byte in descending order as given as follows.

```
.model small
.data
    array  db    12h,11h,21h,9h,19h
.code
    mov ax,@data          ;Initialize data segment
    mov ds,ax
    mov bx,5              ;Initialize pass counter
up1:
    mov si,offset array   ;Initialize memory pointer
    mov cx,4              ;Initialize byte counter
up: mov al,[si]
    cmp al,[si+1]         ;Compare two number
    jnc dn                ;if number>next number then
    xchg al,[si+1]        ;interchange numbers
    xchg al,[si]
dn:  inc si               ;increment memory pointer
    loop up               ;decrement byte counter
                          if ≠ 0 then up
    dec bx                ;decrement pass counter
                          if ≠ 0 then up1
    jnz up1
    ends
    end
```

➢ **Program 15(a)**

Q. 4.3.23 Using the loop instruction of 8086, write an ALP to arrange the contents of memory locations 4000H to 4002H in descending order. Draw a flowchart for the same. (Ref. Program 15(a))

Program

```
    mov ax,1000H          ;Initialize data segment
    mov ds,ax
    mov bx,3              ;Initialize pass counter
                          ;i.e. comparison Counter
up1 :
    mov si,4000H ;Initialize memory pointer
    mov cx,2              ;Initialize word counter
up :
    mov ax,[si]
    cmp ax,[si+1]         ;Compare two numbers
    jnc dn                ;if number > next number
                          ;then go to dn
    xchg ax,[si+21        ;interchange numbers
    xchg ax,[si]
dn:  inc si               ;increment memory pointer
    loop up               ;decrement byte counter
                          ;if ≠ 0 then up
    dec bx                ;decrement pass counter
                          ;if ≠ 0 then up1
    jnz up1
    ends
    end
```

Syllabus Topic : Sorting Numbers in Ascending and Descending Order

4.3.10 Arrange Number in Ascending Order

➔ **(MSBTE - W-14, S-15, W-15, W-16)**

Q. 4.3.24 Write an assembly language program to sort the array of 5 elements in ascending order. Also draw the flowchart for the same. (Ref. sec. 4.3.10) **W-14, 8 Marks**

Q. 4.3.25 Write an ALP for 8086 to sort the array in ascending order. Draw flowchart. [Assume array of size 10]. (Ref. sec. 4.3.10) **S-15, 4 Marks**

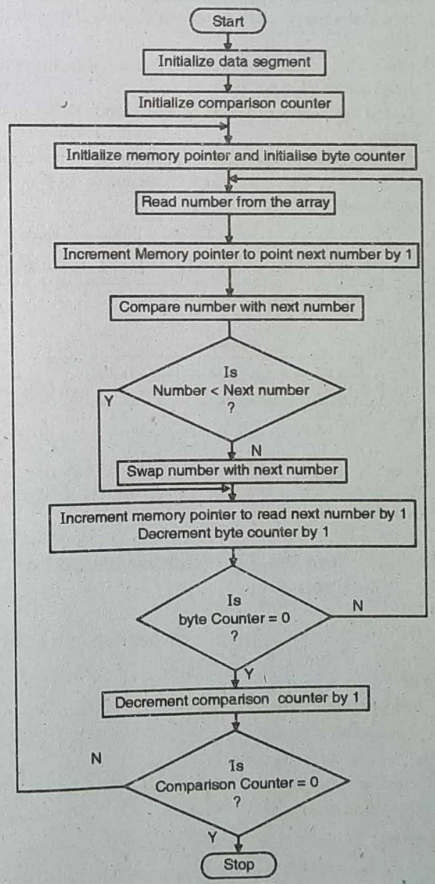
Q. 4.3.26 Write an ALP to sort an array of 10 numbers in Ascending order. (Ref. sec. 4.3.10) **W-15, W-16, 4 Marks**

- To arrange numbers in array in ascending order, the above both i.e. for byte and word program for descending order can be executed by replacing **JNC** instruction with **JC** instruction.
- After execution, we will get the numbers in array in ascending order as given as follows.

Algorithm

1. Initialize data segment.
2. Initialize comparison or pass counter.
3. Initialize memory pointer to read number from array.
4. Initialize word counter.
5. Read numbers from the array.
6. Compare two numbers .
7. If number ≤ to next number then go to step 9.
8. Interchange or swap numbers.
9. Increment memory pointer to read next number from array.
10. Decrement word counter by one.
11. If word counter ≠ 0 then go to step 5.
12. Decrement comparison counter by one.
13. If comparison counter ≠ 0 then go to step 3.
14. Stop.

Flowchart : Refer Flowchart 38.



Flowchart 38

➢ **Program 16**

```
.model small
.data
    array  dw    12h,11h,21h,9h,19h
.code
    mov ax,@data          ;Initialize data segment
    mov ds,ax
    mov bx,5              ;Initialize pass counter
up1:
    mov si,offset array   ;Initialize memory pointer
    mov cx,4              ;Initialize word counter
up:
    mov ax,[si]
    cmp ax,[si+2]         ;Compare two number
    jc dn                 ;if number < next number
                          ; then go to dn
    xchg ax,[si+2]        ;interchange numbers
    xchg ax,[si]
dn:  add si,2             ;increment memory pointer
    loop up               ;decrement word counter
                          ; if ≠ 0 then up
    dec bx                ;decrement pass counter
                          ;if ≠ 0 then up1
    jnz up1
    ends
    end
```

- After the execution of the this program, all five **16 bit numbers** i.e. word will gets arranged in ascending order.
- Same program can be used to arrange **8 bit numbers** i.e. byte in ascending order as given as follows.

```
.model small
.data
    array  db    12h,11h,21h,9h,19h
.code
    mov ax,@data;Initialize data segment
    mov ds,ax
    mov bx,5      ;Initialize pass counter
up1:
    mov si,offset array   ;Initialize memory pointer
    mov cx,4      ;Initialize byte counter
up:
    mov al,[si]
    cmp al,[si+1]         ;Compare two number
    jc dn                 ;if number < next number then
    xchg al,[si+1]        ;interchange numbers
    xchg al,[
```

```
dn:   inc si        ;increment memory pointer
      loop up       ;decrement byte counter if ≠ 0 then up
      dec bx        ;decrement pass counter if ≠ 0 then up1
      jnz up1
      ends
      end
```

➢ **Program 17**

➔ **(MSBTE - S-14)**

Q. 4.3.27 Write an ALP to arrange any array of 10 bytes in an ascending order. Also draw the flow chart for the same. (Ref. sec. 4.3.10 - Program 17) **S-14, 8 Marks**

Flowchart : Please Refer Flowchart 38.

Program

```
.model small
.data
      array   dw   9,6,8,2,6,7,3,4,2,4
.code
      mov ax,@data;Initialize data segment
      mov ds,ax
      mov bx,10              ;Initialize pass counter
up1:
      mov si,offset array    ;Initialize memory pointer
      mov cx,9       ;Initialize word counter
up:
      mov ax,[si]
      cmp ax,[si+2]          ;Compare two number
      jc dn                  ;if number < next number
                             ; then go to dn
      xchg ax,[si+2]         ;interchange numbers
      xchg ax,[si]
dn:   add si,2               ;increment memory pointer
      loop up                ;decrement word counter
                             ;if ≠ 0 then up
      dec bx                 ;decrement pass counter
                             ;if ≠ 0 then up1
      jnz up1
      ends
      end
```

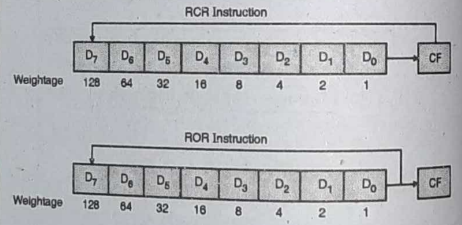
Syllabus Topic : Finding Odd and Even Numbers in the Array

4.3.11 Finding Odd and Even Numbers in the Array

Q. 4.3.28 Write an ALP to check a number to be ODD or EVEN. (Ref. secs. 4.3.11(A) and (B)) **S-14, 4 Marks**

Q. 4.3.29 Write an ALP to compute, whether the number in BL register is even or odd. (Ref. secs. 4.3.11(A) and (B)) **S-17, 4 Marks**

- In 8 bit or 16 bit number, the D_0 bit decides either number is odd or even because the weightage of D_0 bit is 1 i.e. odd value and the weightage of D_1, D_2 ....D_{15} bits are 2, 4, 8... i.e. even value.
- When we add two even or odd numbers, then result is always even, but when we add odd number with even, then result is always odd.
- That's why, when D_0 bit of any number is 1, then that number is odd and if 0 then number is even.
- To test any number for odd or even, check D_0 bit of that number.
- To check D_0 bit of any number, rotate the bits of that number toward left by 1 bit using rotate instruction i.e. ROR or RCR as shown as follows :



- Then D_0 bit goes to the carry flag, hence by checking carry flag, number can be tested for odd or even. It is demonstrated in the following program.

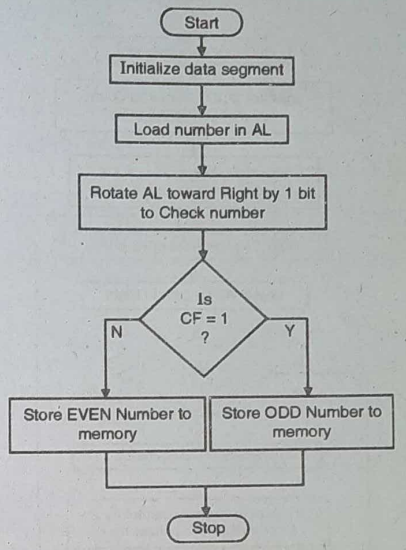
4.3.11(A) Test the 8 Bit Number for Odd or Even

➔ **(MSBTE - S-14, S-17)**

Algorithm

1. Initialize data segment.
2. Load number in resister.
3. Check number is odd or even.
4. If number is odd then store result to odd.
5. Store result to even.
6. Stop.

Flowchart : Refer Flowchart 39.



Flowchart 39

➢ **Program 17**

```
.model small
.data
      num     db   89h
      odd     db   0
      even    db   0
.code
      mov ax,@data  ;Initialize data segment
      mov ds,ax
      mov al,num    ;load number in AL
      ror al,1      ;rotate number by 1 bit toward left
      jnc dn        ;check number odd or even,
      rol al,1      ;if odd, then restore the number
      mov odd,al    ;store in memory variable odd
      jmp exit      ;jump to end the program
dn:   rol al,1      ;else restore number
      mov even,al   ;store in memory variable even.
exit : ends
      end
```

- After the execution of the above program, the given number i.e. 89H is tested and it is stored to memory variable odd.

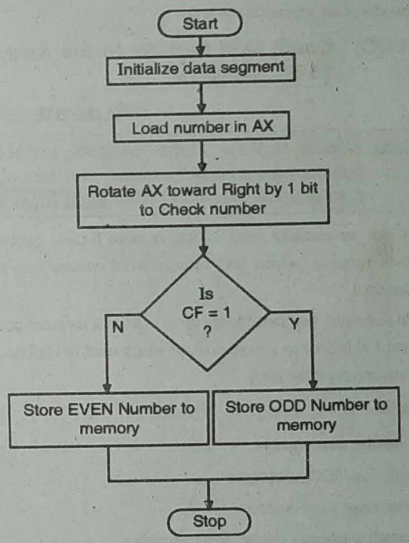
4.3.11(B) Test the 16 Bit Number for Odd or Even

➔ **(MSBTE - S-14, S-17)**

Algorithm

1. Initialize data segment.
2. Load number in resister.
3. Check number is odd or even.
4. If number is odd then store result to odd.
5. Store result to even.
6. Stop.

Flowchart : Refer Flowchart 40.



Flowchart 40

➢ **Program 18**

```
.model small
.data
      num     dw   8988h
      odd     dw   0
      even    dw   0
.code
      mov ax,@data;Initialize data segment
      mov ds,ax
      mov ax,num      ;load number in AX
      ror ax,1        ;rotate number by 1 bit toward left
```

```
    jnc dn          ;check number odd or even,
    rol ax,1        ;if odd, then restore the number
    mov odd,ax      ;store in memory variable odd
    jmp exit        ;jump to end the program
dn: rol ax,1        else restore number
    mov even,ax     ;store in memory variable even.
exit : ends
    end
```

- After the execution of the program 18, the given number i.e. 8988 H is tested and it is stored to memory variable odd.
- So this logic can be used to count how many odd or even numbers are present in the array.

4.3.11(C) Count Odd Number in the Array of 16 Bit Numbers

➔ **(MSBTE - W-15)**

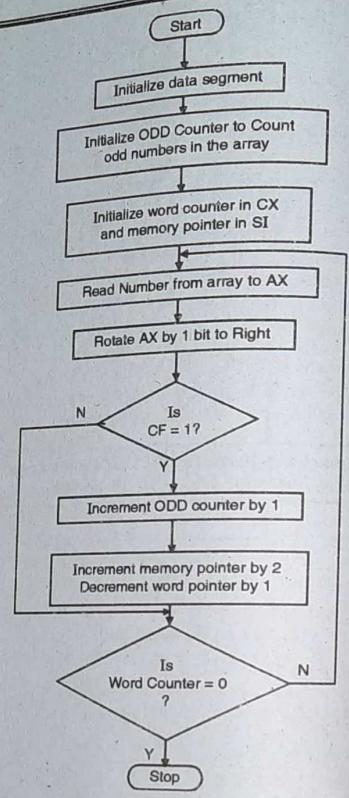
Q. 4.3.30 Write an ALP for 8086 to count the number of odd numbers in array. [Assume array size of 20 number] (Ref. sec. 4.3.11.(C)) W-15, 4 Marks

- When we consider array of byte or word for any operation, then memory pointer and byte or word counter is always required.
- In following program SI register is used as a memory pointer and CX is used as a word counter which used to read desired numbers from the array.

Algorithm

1. Initialize data segment.
2. Initialize ODD_counter to 0.
3. Increment word counter.
4. Initialize memory pointer to read number.
5. Read number.
6. Check number for ODD.
7. If number ≠ ODD then go to step 9.
8. Increment ODD_counter by 1.
9. Increment memory pointer to read next number.
10. Decrement word counter by one.
11. If word counter ≠ 0 then go to step 5.
12. Store result.
13. Stop.

Flowchart : Refer Flowchart 41.



Flowchart 41

➤ **Program 19**

```
.model small
.data
    array   dw   134h,65h,876h,976h,23h
    odd_no  dw   0
.code
    mov ax,@data;Initialize data segment
    mov ds,ax
    mov cx,5        ;Initialize word counter
    mov si,offset array     ;Initialize memory pointer
next:
    mov ax,[si]             ;Read number from array
    ror ax,1
    jnc dn                  ;Check number for odd
    inc odd_no              ;if number is odd
dn:                         ;increment odd counter
    add si,2        ;increment memory pointer
    loop next;decrement word counter
    ends
    end
```

- In above program, array contains two odd numbers.
- Hence after the execution of above program, we will get the result **02h** in memory variable **odd_no.**

➤ **Program 19(a)**

➔ **(MSBTE - S-14, W-16)**

Q. 4.3.31 Write an ALP to transfer block of 10 numbers from one location to another location. (Ref. Program 19(a)) S-14, W-16, 4/8 Marks

Program

```
.model small
.data
    src_arr  dw   1,2,3,4,5,6,7,8,9,10
    dst_arr  dw   5 dup(0) ;Empty array
.code
    mov ax, @data           ;Initialize data segment
    mov ds,ax
    mov cx,10               ;Initialize word counter
    mov si,offset src_arr   ;Initialize memory pointer for
                            ; source
    mov di, offset dst_arr  ;Initialize memory pointer for
up:                         ;destination
    mov ax,[si]             ;read number from source array
    mov [di],ax             ;write number to destination
                            ;array
    add si,2                ;increment source memory
                pointer
    add di,2                ;increment destination
                            memory pointer
    loop next       ;check  word  counter  for  zero,
                            ; if not zero then read next
                            ;number from the array.
    ends
    end
```

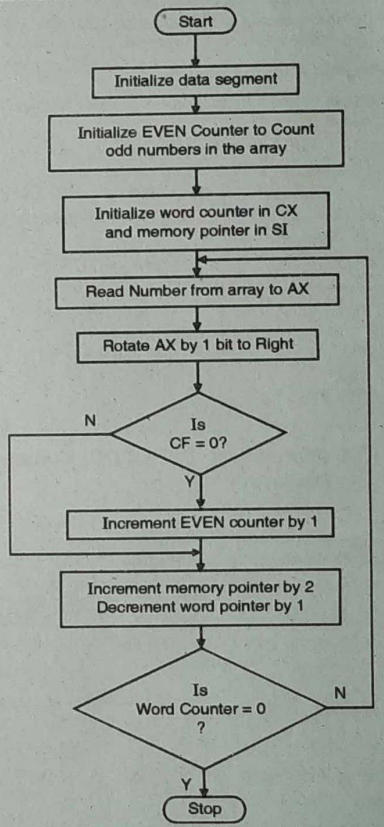
4.3.11(D) Count Even Numbers in the Array of 16 Bit Numbers

- Above program for counting odd numbers in the array can be used to count even numbers in the array by simply replacing **JNC** instruction with **JC** instruction.
- In following program SI register is used as a memory pointer and CX is used as a word counter which used to read desired numbers from the array.

Algorithm

1. Initialize data segment.
2. Initialize EVEN_counter to 0.
3. Increment word counter.
4. Initialize memory pointer to read number.
5. Read number.
6. Check number for EVEN.
7. If number ≠ EVEN then go to step 9.
8. Increment EVEN_counter by 1.
9. Increment memory pointer to read next number.
10. Decrement word counter by one.
11. If word counter ≠ 0 then go to step 5.
12. Store result.
13. Stop.

Flowchart : Refer Flowchart 42.



Flowchart 42

➤ **Program 20**

```
.model small
.data
array  dw    134h,65h,876h,976h,23h
even_no      dw    0
.code
mov ax,@data            ;Initialize data segment
```

```
mov ds,ax
mov cx,5                ;Initialize word counter
mov si,offset array ;Initialize memory pointer
next:
mov ax,[si]             ;Read number from array
ror ax,1            ;Check number for even
jc dn                   ;if number is even
inc even_no             ;increment even counter
dn:
add si,2                ;increment memory pointer
loop next               ;decrement word counter
ends
end
```

– After the execution of the above program, the result is 03 which will be available in memory variable even_no.

➢ **Program 20(a)**

– Program to find number stored at memory location 1D005H is odd or even.

```
Given   :   1D005H = 8 bit Number
Store Result in :   1D006 = if number is even
                    else 1D007 = number is odd
MOV AX, 1D00H       ; Initialize-data segment
MOV DS, AX
MOV AL, [0005H]     ; Load number in AL
MOV BL, AL
ROR AL, 1           ;Check no is odd or even
JC DN
MOV [0006H], BL     ;Store no in memory
      JMP EXIT
DN:  MOV [0007H], BL
EXIT:
```

4.3.11(E) Addition of Only ODD Numbers in the Array

➔ **(MSBTE - W-14)**

Q. 4.3.32 Write as assembly language program to add only odd numbers in the list of following element. 6, 5, 21, 3, 8, 9. (Ref. sec. 4.3.11(E)) W-14, 4 Marks

– Here we must separate the odd numbers from the list and then we can add all the odd numbers.

```
.model small
.data
    arr       db    6,5,21,3,8,9
    arr_odd db 6 dup(0)  ;Array to store odd numbers
    count     dw    0
     sum      db    0      ; result of addition of ODD nos.
.code
     mov ax,@data        ; Initialization of data segment
     mov ds,ax
     mov cx,6            ; Byte Counter = 6 as
                           six numbers in list
     mov si,offset arr   ; Initialize memory pointers
     mov di,offset arr_odd
up:  mov al,[si]
     ror al,1            ; Check number is ODD or not
     jnc dn
     rol al,1        ; If number is ODD then store in
                     ; another  array
     inc count       ; Increment counter by 1
     mov [di],al
     inc di
dn:
     inc si
     loop up         ; decrement byte counter by 1 if not
                     ; zero then go to up
     mov cx,count
     mov si,offset arr_odd
next: mov al,[si]
     add sum,al      ; Add all odd numbers in ODD array
     inc si
     loop next
     ends
     end
```

4.3.11(F) Addition of Only EVEN Numbers in the Array

Here we must separate the even numbers from the list and then we can add all the even numbers.

```
.model small
.data
   arr       db   6,5,21,3,8,9
   arr_even  db   6 dup(0) ;Array for even numbers
   count     dw   0
   sum       db   0       ; result of addition of EVEN nos.
.code
     mov ax,@data      ; Initialization of data segment
     mov ds,ax
     mov cx,6          ; Byte Counter = 6 as
                         six numbers in list
     mov si,offset arr ; Initialize memory pointers
     mov di,offset arr_even
up:  mov al,[si]
     ror al,1          ; Check number is
                         EVEN or not
     jc dn
     rol al,1
                       ;If number is EVEN  then store in
     inc count         ; another array
     mov [di],al       ; Increment counter by 1
     inc di
dn:
     inc si
     loop up
                       ; decrement byte
                       ; counter by 1 if not zero
     mov cx,count      ; then go to up
     mov si,offset arr_even
next: mov al,[si]
     add sum,al        ; Add all even numbers in EVEN array
     inc si
     loop next
     ends
     end
```


Syllabus Topic : Finding Positive and Negative Numbers in Array

4.3.12 Finding Positive and Negative Numbers from the Array

– In 8 bit or 16 bit signed magnitude number, the most significant bit indicate sign of the number i.e. D_7 or D_{15} as shown Fig. 4.3.1(a).

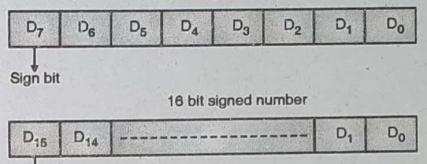


Fig. 4.3.1(a)

– Hence, by checking most significant bit, we can find out a byte or word is positive or negative number.

– Most significant bit i.e. D_7 or D_{15} for byte or word can be checked using either ROL or RCL instruction as given in Fig. 4.3.1(b).

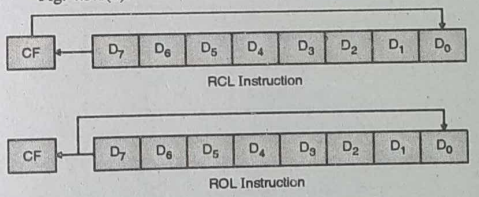


Fig. 4.3.1 (b)

– The program for checking odd or even number can be used by replacing **ROR** or **RCR** instruction with **ROL** or **RCL** instruction to check either number is positive of negative.

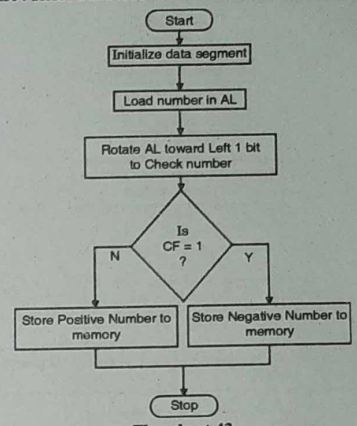
– The following program demonstrates how to number for sign i.e. positive or negative.

4.3.12(A) Test the 8 Bit Number for Positive or Negative

Algorithm

1. Initialize data segment.
2. Load number in resister.
3. Check number is positive or negative.
4. If number is positive then store result to positive and goto step 6.
5. Store result to negative.
6. Stop.

Flowchart : Refer Flowchart 43.



Flowchart 43

➢ **Program 21**

```
.model small
.data
     num     db     89h
     pos     db     0
     neg     db     0
.code
     mov ax,@data     ;Initialize data segment
     mov ds,ax
     mov al,num       ;load number in AL
     rol al,1         ;rotate number by 1 bit toward  left
     jnc dn           ;check number positive or negative
     ror al,1         ;if negative, then restore the number
     mov neg,al       ;store in memory variable neg.
     jmp exit         ;jump to end the program
dn: ror al,1          else restore number
     mov pos,al       ;store in memory variable pos.
exit : ends
     end
```

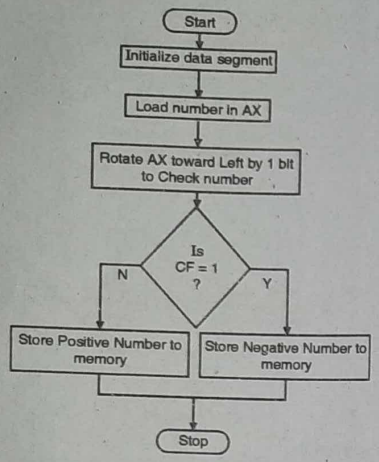
– After the execution of the above program, the given number i.e. 89h is negative as D_7 bit is 1 and it is stored to memory variable **neg.**

4.3.12(B) Test the 16 Bit Number for Positive or Negative

Algorithm

1. Initialize data segment.
2. Load number in resister.
3. Check number is positive or negative.
4. If number is positive then store result to positive and goto step 6.
5. Store result to negative.
6. Stop.

Flowchart : Refer Flowchart 44.



Flowchart 44

➢ **Program 22**

```
.model small
.data
    num   dw  -99h
    pos   dw  0
    neg   dw  0
.code
    mov ax,@data;Initialize data segment
    mov ds,ax
    mov ax,num      ;load number in AL
    rol ax,1        ;rotate number by 1 bit toward left
    jnc dn          ;check number positive or negative
    ror ax,1        ;if negative, then restore the number
    mov neg,ax      ;store in memory variable neg.
    jmp exit        ;jump to end the program
dn: ror ax,1        else restore number
    mov pos,ax      ;store in memory variable pos.
exit: ends
    end
```

- After the execution of the above program, the given number i.e. – 98 H is negative as D_{15} bit is 1 and it is stored to memory variable **neg.**
- So this logic can be used to count how many positive or negative numbers are present in the array.

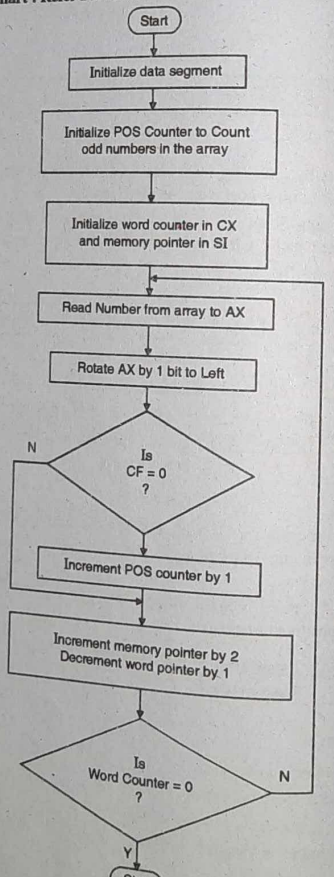
4.3.12(C) Count Positive Number In the Array of 16 Bit Numbers

- When we consider array of byte or word for any operation, then memory pointer and byte of word counter is always required.
- In following program SI register is used as a memory pointer and CX is used as a word counter which used to read desired numbers from the array.

Algorithm

1. Initialize data segment.
2. Initialize POSITIVE_counter to 0.
3. Increment word counter.
4. Initialize memory pointer to read number.
5. Read number.
6. Check number for POSITIVE.
7. If number ≠ POSITIVE then go to step 9.
8. Increment POSITIVE_counter by 1.
9. Increment memory pointer to read next number.
10. Decrement word counter by one.
11. If word counter ≠ 0 then go to step 5.
12. Store result.
13. Stop.

Flowchart : Refer Flowchart 45.



Flowchart 45

➢ **Program 23**

```
.model small
.data
    array   dw   134h,65h,876h,976h,23h
    pos_no  dw   0
.code
    mov ax,@data;Initialize data segment
    mov ds,ax
    mov cx,5        ;Initialize word counter
    mov si,offset array     ;Initialize memory pointer
next:
    mov ax,[si]             ;Read number from array
    rol ax,1                ;Check number for positive
    jc dn                   ;if number is positive
    inc pos_no              ;increment pos counter
dn:
    add si,2                ;increment memory pointer
    loop next       ;decrement word counter
    ends
    end
```

- In above program, array contains three positive numbers i.e. 134h, 65h and 23h as D_{15} bit of these numbers are 0.
- Hence after the execution of above program, we will get the result **03h** in memory variable **pos_no.**

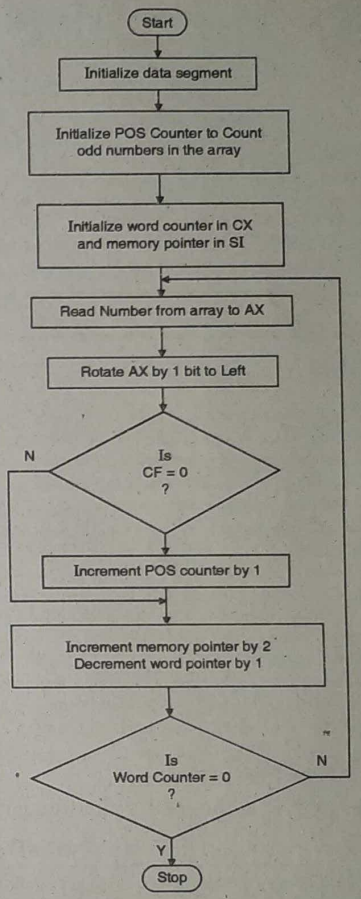
4.3.12(D) Count Negative Numbers In the Array of 16 Bit Numbers

- Above program for counting positive numbers in the array can be used to count negative numbers in the array by simply replacing **JC** instruction with **JNC** instruction.
- In following program SI register is used as a memory pointer and CX is used as a word counter which used to read desired numbers from the array.

Algorithm

1. Initialize data segment.
2. Initialize NEGATIVE_counter to 0.
3. Increment word counter.
4. Initialize memory pointer to read number.
5. Read number.
6. Check number for NEGATIVE.
7. If number ≠ NEGATIVE then go to step 9.
8. Increment NEGATIVE_counter by 1.
9. Increment memory pointer to read next number.
10. Decrement word counter by one.
11. If word counter ≠ 0 then go to step 5.
12. Store result.
13. Stop.

Flowchart : Refer Flowchart 46.



Flowchart 46

➢ **Program 24**

```
.model small
.data
    array    dw   134h,65h,876h,976h,23h
    neg_no   dw   0
.code
    mov ax,@data;Initialize data segment
    mov ds,ax
    mov cx,5        ;Initialize word counter
    mov si,offset array     ;Initialize memory pointer
next:
    mov ax,[si]             ;Read number from array
    rol ax,1                ;Check number for negative
```

```
    jnc dn          ;if number is negative
    inc neg_no      ;increment neg counter
dn :
    add si,2        ;increment memory pointer
    loop next       ;decrement word counter
    ends
    end
```

- In program 24, array contains two positive numbers i.e. 876h and 976h as D_{15} bit of these numbers are 1.
- Hence after the execution of above program, we will get the result **02h** in memory variable **neg_no.**

Syllabus Topic : Block Transfer

4.3.13 Block Transfer

- Assume a block of data on N number is stored in a memory.
- Now this block of N numbers is to be moved from source locations to other part of memory i.e. destination locations.
- If the number of bytes or words N is 5, then we will have to initialize this as byte counter or word counter in CX register.
- Then two memory pointers are required to point source block and destination block, hence we can use SI and DI registers respectively as source and destination memory pointers.
- Then block can be transfer from source to destination either using string instruction i.e. MOVS/MOVSB/MOVSW or without using string instruction such as simple MOV instruction.
- Two array must be declared in the array where in one array actual numbers are stored and another array must be empty.
- To declare empty array, we can use **DUP** directive.
- For example, 5 dup(0) statement allocates five memory location and initialize them with 0.

4.3.13(A) Without using String Instructions

➔ **(MSBTE - S-17)**

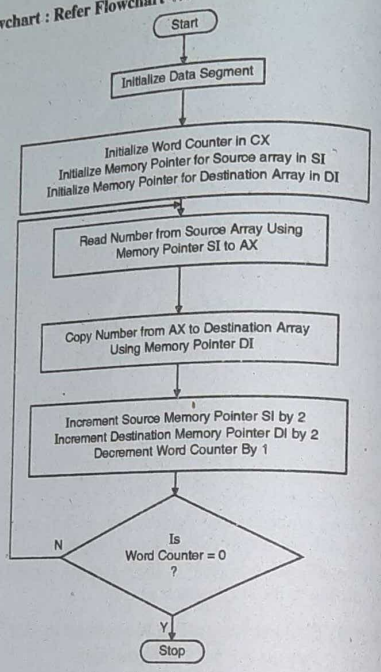
Q. 4.3.33 Write an ALP to transfer 10 bytes of data from one memory location to another. Also draw the flow chart for the same.
(Ref. sec. 4.3.13(A)) **S-17, 8 Marks**

- First we will see the program for block transfer ***without using string instruction*** i.e. using simple **MOV** instruction.
- Following program demonstrate, the block transfer using simple MOV instruction.

Algorithm

1. Initialize data segment
2. Initialize word counter.
3. Initialize memory pointers for source and destination array.
4. Read number from source array.
5. Copy it to destination array.
6. Increment memory pointers for source and destination array for next number.
7. Decrement word counter by one.
8. If word counter ≠ 0 then go to step 4.
9. Stop.

Flowchart : Refer Flowchart 47.



Flowchart 47

➢ **Program 25**

```
.model small
.data
    src_arr  dw  1234h, 4321h, 7894h,9658h,45ABh
    dst_arr  dw  5 dup(0)        ;Empty array
.code
    mov ax, @data                ;Initialize data segment
    mov ds,ax
    mov cx,5                     ;Initialize word counter
    mov si, offset src_arr       ;Initialize memory
                                 pointer for source
    mov di, offset dst_arr ;Initialize memory pointer for
up:                        ; destination
    mov ax,[si]     ;read number from source array
    mov [di],ax     ;write number to destination array
    add si,2        ;increment source memory pointer
    add di,2        ;increment destination memory ;pointer
    loop next       ;check word counter for zero,
                    ; if not zero then read  next
    ends            ;number from the array.
    end
```


4.3.13(B) Using String Instructions

➔ **(MSBTE - W-17, S-18)**

Q. 4.3.34 Write ALP and draw flow chart to perform block transfer without using string instruction.
(Ref. sec. 4.3.13(B)) **W-17, S-18, 4/8 Marks**

- Now, we will see the program for block transfer **using string instruction** i.e. using **MOVSW** instruction.
- For **MOVSW** instruction the default memory pointer for source and destination blocks are **DS:SI** and **ES:DI** respectively.
- Following program demonstrate, the block transfer using simple **MOVSW** instruction.

Algorithm

1. Initialize data and extra segment i.e. **DS** and **ES.**
2. Initialize word counter.
3. Initialize memory pointers for source and destination array.
4. Read number from source array.
5. Copy it to destination array.
6. Increment memory pointers for source and destination array for next number.
7. Decrement word counter by one.
8. If word counter ≠ 0 then go to step 4.
9. Stop.

Flowchart : Refer Flowchart 47.

➢ **Program 26**

```
.model small
.data
    src_arr  dw  1234h,4321h,7894h,9658h,45ABh
    dst_arr  dw  5 dup(0)     ;Empty array
.code
    mov ax, @data
    mov ds,ax              ;Initialize data segment
    mov es,ax              ;Initialize extra segment
    mov cx,5               ;Initialize word counter
    mov si, offset src_arr ;Initialize memory pointer for
                           ;source
    mov di, offset dst_arr ; Initialize memory pointer
                           ; for destination
up : movsw                 ;Transfer word from source to
                           ; destination
    loop up
    ends
    end
```

➢ **Program 26(a)**

- Program to transfer 10 byte data from base of data segment to base of extra segment. Data segment base address is 2C000H. Extra segment base address is DE000H.
- Refer Algorithm and flowchart of program 25

```
MOV AX, 2C00H          ; Initialize data segment
MOV DS, AX
MOV AX, DE00H          ; Initialize Extra segment
MOV ES, AX
MOV SI, 0000H          ;Initialize memory pointers
MOV DI, 0000H
MOV CX, 000AH          ; Initialize counter
UP: MOV AL, [SI]       ; Transfer data from source
MOV ES:[DI], AL        ; to destination
INC SI                 ; Increment memory pointers
INC DI
LOOP UP                ;Check counter≠0 then up
ENDS
END
```

➢ **Program 26(b)**

Using string instruction, write an ALP to transfer a block of 1 Kb of data stored at location 1000H onward to location 4000H onwards in the data segment

```
MOV AX, 1000H          ; Initialize data segment
MOV DS, AX
MOV AX, 4000H          ; Initialize Extra segment
MOV ES, AX
MOV SI, 0000H
MOV DI, 0000H
MOV CX, 03FFH
UP: MOVSB
LOOP UP
```

➢ **Program 26(c) : Overlapping Block Transfer**

➔ **(MSBTE - S-15)**

Q. 4.3.35 Write an algorithm to transfer block of data from the source address to destination address and vice versa [Overlapping block transfer].
(Ref. Program 26(c)) **S-15, 4 Marks**

Algorithm

1. Initialize data segment
2. Initialize word counter.
3. Initialize memory pointers for source and destination array at last number
4. Read number from source array.
5. Copy it to destination array.
6. Decrement memory pointers for source and destination array for next number.
7. Decrement word counter by one.
8. If word counter ≠ 0 then go to step 4.
9. Stop.

Program

```
.model small
.data
src_arr   dw   1234h, 4321h, 7894h,9658h,45ABh
dst_arr   dw   5 dup(0)  ;Empty array
.code
mov ax, @data              ;Initialize data segment
mov ds,ax
mov cx,5                   ;Initialize word counter
mov si, offset src_arr     ;Initialize memory
                           ;pointer for source
mov di, offset dst_arr     ;Initialize memory
                           pointer for destination
add si,4                   ; set memory pointers
                           to last number
add di,4                   ; for source and destination
up:
mov ax,[si]     ;read number from source array
mov [di],ax     ;write number to destination array
sub si,2        ;decrement source memory pointer
sub di,2        ;decrement destination memory pointer
loop next       ;check word counter for zero,
                ; if not zero then read next
                ;number from the array.
ends
end
```

➤ **Program 26(d)**

➔ **(MSBTE - W-15)**

Q. 4.3.36 Write an ALP to transfer a block of 50 numbers from 20000 H to 30000 H. (Ref. Program 16(d)) **W-15, 4 Marks**

```
CODE SEGMENT
ASSUME CS: CODE
START:    MOV AX, 2000H    ; Initialize data segments
          MOV DS, AX
          MOV AX, 3000H    ; Initialize extra segments
          MOV ES, AX
          MOV SI, 0000H    ;Initialize memory pointers
          MOV DI, 0000H
          MOV CX, 0032H    ; Initialize byte
                             counter 50
UP:       MOV AL,[SI]      ;Transfer data from
                           source to destination
          MOV ES: [DI], AL
          INC SI           ; increment memory
                           pointers by 1
          INC DI
          LOOP UP          ; Check byte counter if
                           not zero then loop up
          MOV AH, 4CH      ; Terminate the program
          INT 21H
CODE ENDS
          END START
```

4.3.14 Comparison of Two Strings

➔ **(MSBTE - S-14)**

Q. 4.3.37 Write an ALP to compare two string of 10 byte each. (Ref. sec. 4.3.14) **S-14, 4 Marks**

- The string consists of either numbers or characters. In assembly, the string must be declare in quotes i.e. ' ' and string must end with '$' sign.
- The data type of the string is always byte because assembler store ASCII value of every character of string in memory, as ASCII codes are 8 bit.

For Example

```
name   db   'Computer$'
dept   db   'Information technology$'
```

- Assembler stores string characters in memory at consecutive memory locations.
- Hence to perform any string related operation such as comparison, length, reverse etc., the memory pointer and byte counter is required as we have seen in block transfer program.
- Now, we will see how to compare two strings either without using or using string instruction.
- For comparison, the string instruction is CMPS or CMPSB or CMPSW.
- But, normally we use CMPSB instruction for string comparison as data type of string is byte if it is specified in a quotes. Simple CMP instruction also can be used to compare two strings.

4.3.14(A) Without using String Instructions

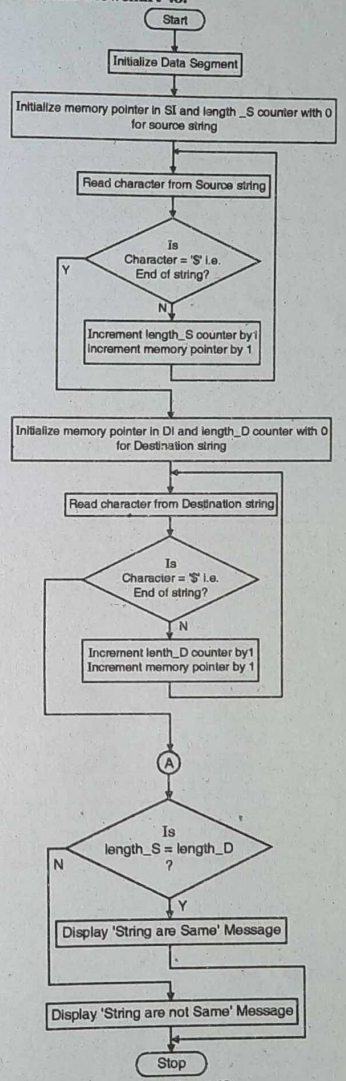
- First we will see the program for comparison of two strings **without using string instruction** i.e. using simple CMP instruction.
- Following program demonstrate, the using simple CMP instruction.

1. Comparison of string by checking length of both the string.

Algorithm

1. Initialize data segment.
2. Find the length of source string.
3. Find the length of destination string.
4. Compare length of both the strings.
5. If length of both string are not same then go to step 8.
6. Display message 'Strings are same'.
7. Stop.
8. Display message 'String are not same'.
9. Stop.

Flowchart : Refer Flowchart 48.



Flowchart 48

➤ **Program 27**

```
.model small
.data
    str_s     db    'COMPUTER$'
    str_d     db    'computer$'
    count_s   db    0
    count_d   db    0
    msg1      db    'Strings are Same$'
    msg2      db    'Strings are Not Same$'
.code
    mov ax,@data        ;Initialize data segment
    mov ds,ax
;----------Count length of the sources string-----------
    mov si,offset str_s ;Initialize memory pointer for
                        ;source string
next:  mov al,[si]      ;read character from the source
                        ;string
    cmp al,'$'          ;compare with $
    je exit             ;if equal then go to exit to read
                    ;destination ;string
    inc si              ;else increment memory pointer
    inc count_s         ;increment counter to count
                        length of string
    jmp next            ;jump to read next character
    exit:
;---------Count length of the Destination string----------
    mov si,offset str_d     ;Initialize memory ptr for
                            ;destination string
next1:  mov al,[si]  ;read character from the
                     ;destination string
    cmp al,'$'       ;compare with $
    je exit1         ;if equal then go to exit1 to
                     ;compare
                     ;length of ;both strings
    inc si           ;else increment memory pointer
    inc count_d      ;increment counter to count
                     length of string
    jmp next1        ;jump to read next character exit1:
;--------Compare length of both the string ----------
    mov al,count_s
    cmp al,count_d
    jne exit2        ;If length of both strings are not
                     ;same then go to exit2 else compare
                     ;strings byte by byte
    mov ah,09h       ;Display string are same message
    lea dx,msg1
    int 21h
    jmp exit3
exit2:
    mov ah,09h       ;Display string are not same message
    lea dx,msg2
    int 21h
exit3:
    mov ah,4ch       ;Terminate program & Exit to DOS
    int 21h
    ends
    end
```

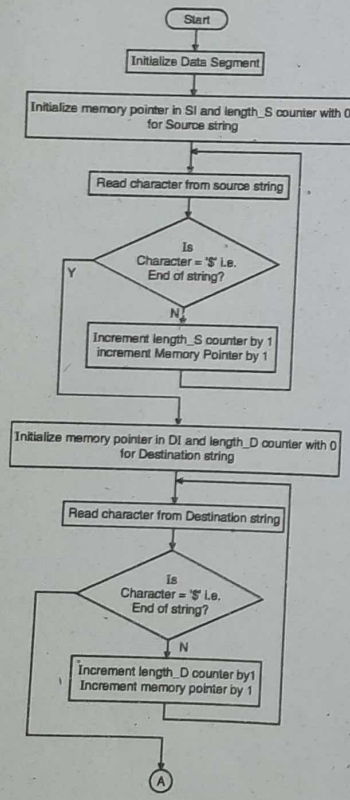
- After the execution of the above program, the message **'String are same'** will displayed on the screen because length of both the strings are same.

2. Comparison of string by checking length and character with case.

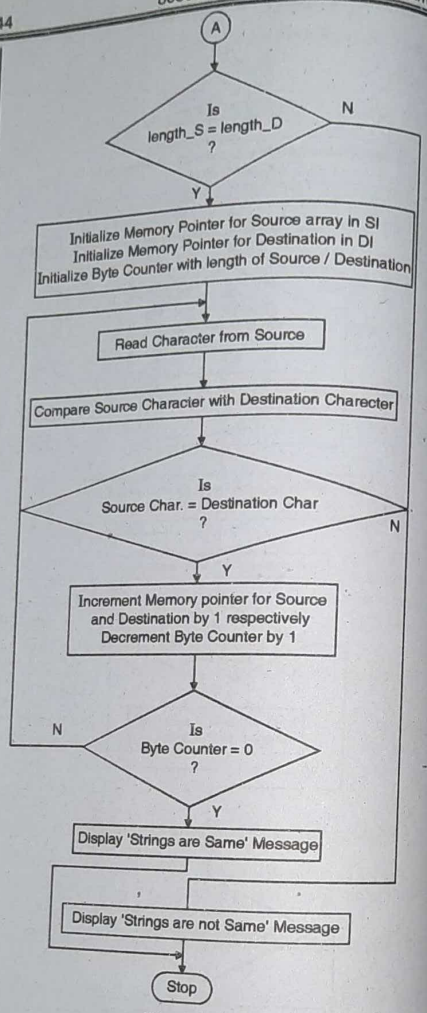
Algorithm

1. Initialize data segment.
2. Find the length of source string.
3. Find the length of destination string.
4. Compare length of both the strings.
5. If length of both string are not same then go to step 10.
6. Compare string character by character.
7. If characters of both the strings are not same then go to step 10.
8. Display message 'Strings are same'.
9. Stop.
10. Display message 'String are not same'.
11. Stop.

Flowchart : Refer Flowchart 49.



Flowchart 49 cont.....



Flowchart 49

➢ **Program 28**

```
.model small
.data
    str_s     db    'COMPUTER$'
    str_d     db    'computer$'
    count_s db  0
    count_d   db    0
    msg1      db    'Strings are Same$'
    msg2      db    'Strings are Not Same$'
.code
    mov ax,@data      ;Initialize data segment
    mov ds,ax

;----------Count length of the sources string----------
```



```
    mov si,offset str_s     ;Initialize memory pointer for
                            ; source string
next:  mov al,[si]          ;read character from the source
                            ;string
       cmp al,'$'           ;compare with $
       je exit              ;if equal then go to exit to read
                            destination string
       inc si          ;else increment memory pointer
    inc count_s        ;increment counter to count
                            ; length of string
       jmp next             ;jump to read next character
exit:
;--------Count length of the Destination string----------
       mov si,offset str_d  ;Initialize memory ptr for
                            ; destination string
next1: mov al,[si] ;read character from the
                            ; destination string
       cmp al,'$'           ;compare with $
       je exit1             ;if equal then go to exit1 to
                            ; compare
                            ;length of ;both strings
       inc si               ;else increment memory pointer
       inc count_d          ;increment counter to count
                            ; length of string
       jmp next1            ;jump to read next character
exit1:
    ;----------Compare length of both the string ----------
    mov al,count_s
    cmp al,count_d
    jne exit2               ;If length of both strings are not
                            ;same then ;go to exit2 else
                            compare
                            ;strings byte by byte
    mov si,offset str_s     ;Initialize memory pointer for
                            ;source string
    mov di,offset str_d     ;Initialize memory pointer for
                            ;destination ;string
up: mov al,[si]             ;read character from source
                            ;string
    cmp al,[di]             ;compare it with character of
                            ;destination ;string
    jne exit2               ;if character are not equal then
                            ; go to exit2
    inc si                  ;else increment source memory
                            ;pointer
    inc di                  ;increment destination memory
                            ;pointer
    dec count_s             ;decrement byte counter
    jnz up                  ;if byte counter ≠ 0 then go to
                            ; up for next characters
                            ; comparison
    mov ah,09h              ;Display string are same message
    lea dx,msg1
    int 21h
    jmp exit3
exit2:
    mov ah,09h  ;Display string are not same message
    lea dx,msg2
    int 21h
exit3:
    mov ah,4ch  ;Terminate program & Exit to DOS
    int 21h
    ends
    end
```

- Read above program carefully, you will find very lengthy.
- But it covers all aspects or you can say that possibilities of string comparison.
- If the lengths of strings are same, it does not mean that strings are same.
- Although the lengths of both string are same, but the characters and their case i.e. lower or upper case, in strings may be different.
- So, above program satisfy all the above aspects or possibilities.
- Again, at the end of the program, you will find two DOS functions i.e. 09h and 4ch of interrupt 21h.
- The function **09h** is used to display string terminated with $ on the standard output device i.e. monitor.
- To call this function **ah** must be loaded with function code **09h** and **dx** must be loaded with **offset address of string.**
- The another function **4ch** is used to terminate program with return code 0 indicating successful execution and transfer program control back to operating system.
- So, **ah** must be loaded with the function code **4ch** to call it.

➢ **Program 28(a)**

```
.model small
.data
    str_s  db  '.....$'  ;declare 50 character
                         ; source string
    str_d  db  '.....$'  ;declare 50 character
                         ;destination string
    msg1   db  'Strings are Same$'
    msg2   db  'Strings are Not Same$'
.code
    mov ax,@data;Initialize data segment
    mov ds,ax
    mov cx,50
    mov si,offset str_s     ;Initialize memory pointer for
                            ;source string
    mov di,offset str_d ;Initialize memory pointer for
                            ;destination string
up :            ;Compare character of source string with
    cmpsw       ; character of destination string
    jnz exit2   ;if character are not equal then
                ;go to exit2
    loop up     ;decrement byte counter if byte
                ;counter ≠ 0 then go to up for next
                ;character comparison
```

```
    mov ah,09h    ;Display string are same message
    lea dx,msg1
    int 21h
    jmp exit3
exit2:
    mov ah,09h    ;Display string are not same message
    lea dx,msg2
    int 21h
exit3:
    mov ah,4ch    ;Terminate program & Exit to DOS
    int 21h
    ends
    end
```

4.3.14(B) Using String Instructions

- Now, we will see the program for string comparison ***using string instruction*** i.e. using **CMPSB** instruction if string is terminated with $ sign.
- For **CMPSB** instruction the default memory pointer for source and destination blocks are **DS:SI** and **ES:DI** respectively.
- Following program demonstrate, the comparison of two strings using **CMPSB** instruction.

Algorithm

1. Initialize data and extra segment i.e. DS and ES.
2. Find the length of source string.
3. Find the length of destination string.
4. Compare length of both the strings.
5. If length of both string are not same then go to step 10.
6. Compare string character by character.
7. If characters of both the strings are not same then go to step 10.
8. Display message 'Strings are same'.
9. Stop.
10. Display message 'String are not same'.

Flowchart : Refer Flowchart 49.

➢ **Program 29**

```
.model small
.data
    str_s    db   'COMPUTER$'
    str_d    db   'computer$'
    count_s  db   0
    count_d  db   0
    msg1     db   'Strings are Same$'
    msg2     db   'Strings are Not Same$'
.code
    mov ax,@data;Initialize data segment
    mov ds,ax
    mov es,ax              ;Initialize extra segment
;--------Count length of the sources string----------
    mov si,offset str_s    ;Initialize memory pointer for
                           ;source string
next:  mov al,[si]         ;read character from the
                           ;source string
    cmp al,'$'             ;compare with $
    je exit                ;if equal then go to exit to read
                           ;destination string
    inc si                 ;else increment memory pointer
    inc count_s            ;increment counter to count
                           ;length of string
    jmp next               ;jump to read next character
exit:
;--------Count length of the Destination string----------
    mov si,offset str_d    ;Initialize memory ptr for
                           ;destination string
next1:  mov al,[si]        ;read character from the
                           ;destination string
    cmp al,'$'             ;compare with $
    je exit1               ;if equal then go to exit1
                           ;to compare length of both strings
    inc si                 ;else increment memory pointer
    inc count_d            ;increment counter to count
                           ;length of string
    jmp next1              ;jump to read next character
exit1:
    ;--------Compare length of both the string ----------
    mov al,count_s
    cmp al,count_d
    jne exit2              ;If length of both strings are not
                           ;same then go to exit2 else
                           ; compare strings byte by byte
    cld                    ;clear direction flag
    mov ch,0
    mov cl,count_s         ;Initialize byte counter i.e.
                           ;length of string
    mov si,offset str_s    ;Initialize memory pointer for
                           ;source string
    mov di,offset str_d    ;Initialize memory pointer for
                           ;destination string
up :                       ;Compare character of source
    cmpsb                  ;string with
                           ;character of destination
    jnz exit2              ;string
                           ;if character are not equal then
    loop up                ;go to exit2
                           ;decrement byte counter if byte
                           ;counter ≠ 0 then go to up for
    mov ah,09h             ;nextcharacter comparison
    lea dx,msg1            ;Display string are same message
    int 21h
    jmp exit3
```



```
exit2:
    mov ah,09h             ;Display string are not same
                           ;message
    lea dx,msg2
    int 21h
exit3:
    mov ah,4ch             ;Terminate program & Exit to DOS
    int 21h
    ends
    end
```

- After the execution of the above program, the message **'String are not same'** will be displayed on the screen.
- The lengths of both the strings are same, but the cases of characters in both the string are different. Hence strings are not same.

Syllabus Topic : String Operation Reverse

4.3.15 Display String in Reverse Order

➔ **(MSBTE - W-16, S-17)**

Q. 4.3.38 Write an ALP to reverse a string of 8 characters. (Ref. sec. 4.3.15) **W-16, 4 Marks**

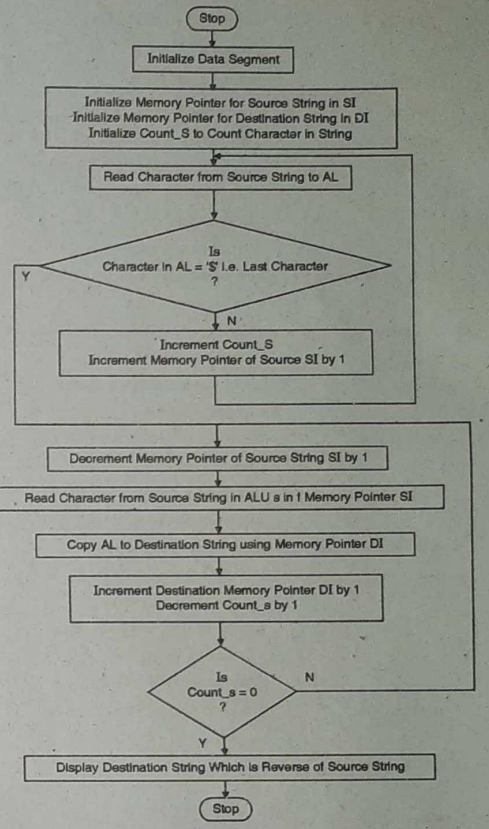
Q. 4.3.39 Write an ALP to reverse the string. (Ref. sec. 4.3.15) **S-17, 4 Marks**

- In C language, **strrev** function is used to perform string reverse operation.
- In assembly language, we can perform same operation.
- So, memory pointer and length counter should be initialized to read string and then copy string in another blank string variable in reverse order.
- To reverse the string, first find out the length of the source string, then add this value to memory pointer register to point last character of the source string.
- Then copy last character from source string to first character position of destination blank string.
- Perform this operation continuously till first character of the source string gets transfer to destination string by decrementing memory pointer for source string and incrementing memory pointer for destination string.
- The following program performs string reverse operation and displays it on screen.

Algorithm

1. Initialize data segment.
2. Find length of source string.
3. Copy source string to destination string in reverse order.
4. Display both source and destination string.
5. Stop.

Flowchart : Refer Flowchart 50.



Flowchart 50

➢ **Program 30**

```
.model small
.data
    str_s   db  'COMPUTER DEPARTMENT$'
    str_d   db  50 dup('$')
    msg1    db  10,13,'The source String      :$',10,13
    msg2    db  10,13,'The String After Reverse  :$',10,13
    count   db  0
.code
    mov ax,@data          ;initialize data segment
    mov ds,ax             ;Calculate length of source string
    mov si,offset str_s   ;Initialize memory pointer for
                          ;source string
next: mov al,[si]         ;read first character from
                          ;source string
```

```
    cmp al,'$'          ;check for end of string,
                        ;if yes then exit
    je exit             ;else
    inc si              ;Increment memory pointer
    inc count           ;increment length counter
    jmp next            ;jump to read next character
exit :                  ;Copy Source String to
                        ;Destination
                        ;string in reverse order
    mov di,offset str_d ;initialize memory pointer for
                        ;destination string
up :  dec si            ;decrement memory pointer for
                        ;source string
    mov al,[si]         ;read character from source str
                        in reverse order
    mov [di],al         ;copy it to destination string in
                        ;forward order
    inc di              ;increment memory pointer for
                        ;destination string
    dec count           ;decrement length counter
    jnz up              ;if length counter ≠ 0 then
                        ;jump up to copy next
                        ;character display both string
                        ;On screen using
                        ;09h function of INT 21h
    mov ah,09h          ;Display Source string on the
                        ;screen
    lea dx, msg1
    int 21h
    mov ah,09h
    lea dx,str_s
    int 21h
    mov ah,09h  ;Display Destination string on screen
    lea dx, msg2
    int 21h
    mov ah,09h
    lea dx, str_d
    int 21h
    mov ah,4ch          ;Exit to DOS
    int 21h
    ends
    end
```

The output of the program

```
G:\tasm\str>strrev
The source String       : COMPUTER DEPARTMENT
The String After Reverse : TNEMTRAPED RETUPMOC
```

Syllabus Topic : String Operation Length

4.3.16 Find Length of String

➔ **(MSBTE - W-14, S-16, W-17)**

Q. 4.3.40 Write an assembly language program to find the length of a string using 8086. (Ref. sec. 4.3.16) **W-14, W-17, 4 Marks**

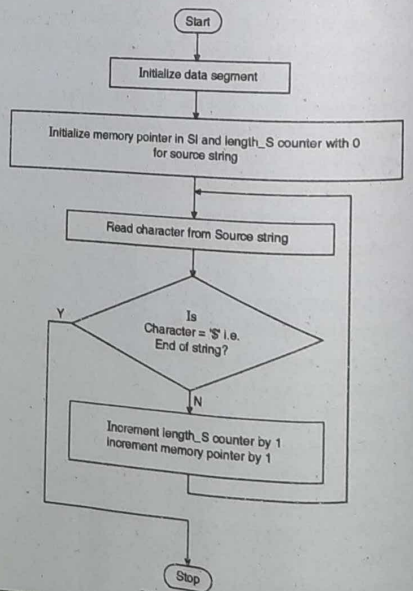
Q. 4.3.41 Write an assembly language program to find length of string. (Ref. sec. 4.3.16 - Program 31) **S-16, 4 Marks**

- To find the length of the string, take one length counter and initialize memory pointer to read character from the string.
- Read character from the array and compare it with '$' which indicate end of the string.
- If the character is not '$' then increment length counter else stop reading character from the string.

Algorithm

1. Initialize data segment.
2. Initialize length counter.
3. Initialize memory pointer to read character from string.
4. Read character from the string.
5. If character is '$' then go to step 9.
6. Increment length counter.
7. Increment memory pointer to next character.
8. Go to step 4.
9. Stop.

Flowchart : Refer Flowchart 51.



Flowchart 51

➢ **Program 31**

```
.model small
.data
    str_s   db   'COMPUTER$'
    length  db   0
.code
    mov ax,@data        ;Initialize data segment
    mov ds,ax
    mov si,offset str_s ;initialize memory pointer
next: mov al,[si]       ;read character
    cmp al,'$'          ;check for end of string
    je exit             ; if not end of string then
    inc si              ;increment memory pointer
    inc length          ;increment length counter
    jmp next            ;jump to read next character
exit:
    ends
    end
```

- The output of the above program can be seen by debugging above program.
- Hence, after the execution of the program, the value of the memory variable ***length*** will be *8* which is nothing but length of the string.

Syllabus Topic : String Operation Concatenation

4.3.17 Concatenation of Two Strings

➔ **(MSBTE - S-15)**

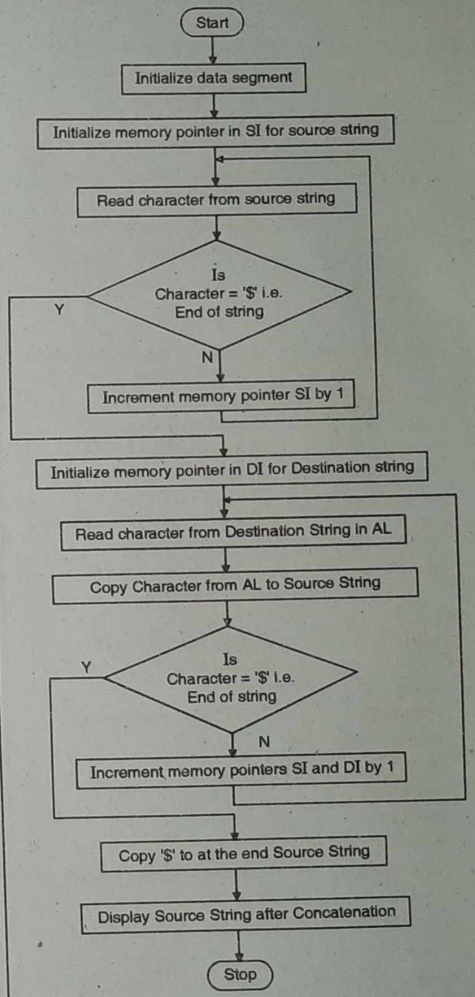
Q. 4.3.42 Write an ALP concatenate two strings with algorithm
String 1 : "Maharashtra Board"
String 2 : "of Technical Education".
(Ref. sec. 4.3.17) **S-15, 8 Marks**

- The concatenation of two strings means merging of second string in first string.
- For example suppose, 'Computer' and 'Department' are two separate strings, after concatenation string will become 'Computer Department'.
- The following program performs the concatenation operation.

Algorithm

1. Initialize data segment.
2. Initialize memory pointers for source and destination.
3. Move memory pointer of source string to end of string.
4. Copy characters from destination string to source string.
5. Stop.

Flowchart : Refer Flowchart 52.



Flowchart 52

➢ **Program 32**

```
.model small
.data
    str_s   db   'COMPUTER ENGINEERING$'
    str_d   db   '_INFORMATION TECHNOLOGY$'
    msg1    db   'After Concatenation....:: $'
.code
    mov ax,@data;Initialize data segment
```

```
    mov ds,ax           ;move memory pointer to the last
                        ;character of the source string
    mov si,offset str_s ;Initialize memory pointer
                        ;for source string
next:
    mov al,[si]
    cmp al,'$'          ;Is memory pointer is at last
                        ;character
    je exit             ;if yes then jump to
                        concatenate string
    inc si
    jmp next
exit :
    mov di,offset str_d ;initialize memory pointer for
                        ;destination string
up : mov al,[di]        ;read character
    cmp al,'$'          ;check end of string
    je exit1            ;if yes then exit
    mov [si],al         ;else copy character to source
                        ;string
    inc si              ;increment memory pointers for
    inc di              ;source and destination
    jmp up              ;repeat process till end of
                        ; destination string
exit1:
    mov al,'$'          ;end the source string by
                        inserting $
    mov [si],al
    mov ah,09h          ;display concatenate strings
    lea dx, msg1
    int 21h
    mov ah,09h
    lea dx,str_d
    int 21h
     mov ah,4ch         ;Terminate the program & exit to
                        ;DOS
      int 21h
      ends
      end
```

The output of the above program

```
After concatenation :   COMPUTER
                        ENGINEERING_INFORMATION
                        TECHNOLOGY.
```

4.3.18 Convert Lower Case String to Upper Case

- If we see the ASCII character set of all the alphabets we will find that there is a difference of 20H between lower case and upper case.
- For example, the ASCII code of upper case character 'A' is 41 and the ASCII code of lower character 'a"is 61.
- So to convert character from lower case to upper case, simply subtract 20H from the ASCII of lower case character and to convert character from upper case to lower case, simply add 20H to the ASCII of upper case character

Algorithm

Step 1 : Initialize data segment
Step 2 : Initialize memory pointers for source and destination
Step 3 : Read character from source memory
Step 4 : If end of string then go to step 9
Step 5 : Convert from lower to upper case
Step 6 : Store into destination memory
Step 7 : Increment memory pointers by 1
Step 8 : Go to Step 3
Step 9 : Stop

Program

```
.model small
.data
      str_l   db   'computer$'
      str_u   db   20 dup('$')
.code
      mov ax,@data      ;Initialise data segment
      mov ds,ax
      mov si,offset str_l   ;initialize memory pointers
      mov di,offset str_u,
next:. mov al,[si]      ;Read character from array
      cmp al,'$'        ;If end of string then exit
      je exit
      sub al,20H        ;Convert to upper case
      mov [di],al       ;store into memory
      inc si            ;increment memory pointers
      inc di
      jmp next          ; goto next charecter
exit:
      ends
      end
```

4.3.19 Convert Upper Case String to Lower Case

- In case to convert from upper case to lower case simply replace the instruction SUB AL,20H by ADD AL,20H

Algorithm

Step 1 : Initialize data segment
Step 2 : Initialize memory pointers for source and destination
Step 3 : Read character from source memory
Step 4 : If end of string then go to step 9
Step 5 : Convert from Upper to lower case
Step 6 : Store into destination memory

Step 7 : Increment memory pointers by 1
Step 8 : Go to Step 3
Step 9 : Stop

Program

```
.model small
.data
      str_l   db   'computer$'
      str_u   db   20 dup('$')
.code
      mov ax,@data      ;Initialise data segment
      mov ds,ax
      mov si,offset str_l   ;initialize memory pointers
      mov di,offset str_u
next:  mov al,[si]      ;Read character from array
      cmp al,'$'        ;If end of string then exit
      je exit
      sub al,20H        ;Convert to lower case
      mov [di],al       ;store into memory
      inc si            ;increment memory pointers
      inc di
      jmp next          ; goto next charecter
exit:
      ends
      end
```

4.3.20 Convert BCD Number to Hexadecimal

➔ **(MSBTE - W-16, S-18)**

Q. 4.3.43 Write an ALP to convert BCD to HEX. (Ref. sec.4.3.20) W-16, S-18, 4 Marks

Q. 4.3.44 Write algorithms and draw flow chart to convert BCD no. to Hex numbers. Also write the assembly language program for 8086. (Ref. sec. 4.3.20)

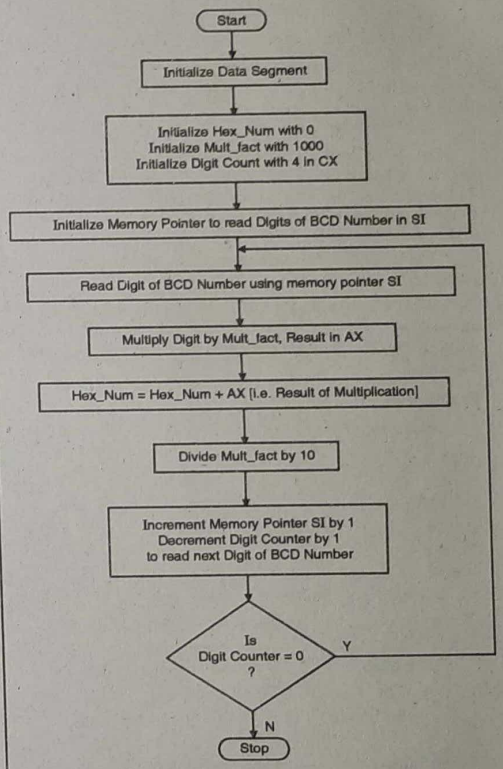
- The conversion method is based on the fact that BCD number is in base 10 and the computer performs arithmetic in base 2.
- Here is the procedure.
  1. Start with the rightmost byte of the BCD number and process from right to left.
  2. Multiply the first rightmost BCD digit by 1, the second rightmost digit by 10 (0AH), the third by 100 (64H) and so on, and sum the products.

For example, convert BCD number 1234 to binary [Hexadecimal].

Decimal		Hexadecimal	
Step	Product	Step	Product
4 × 1	= 4	4 × 01H	= 4H
3 × 10	= 30	3 × 0AH	= 1EH
2 × 100	= 200	2 × 64H	= C8H
1 × 1000	= 1000	1 × 3E8H	= 3E8H
Total :	1234	Total :	04D2H

The sum 04D2H is equal to decimal number 1234. We can implement above method in following program.

Flowchart : Refer Flowchart 53.



Flowchart 53

➢ **Program 33**

```
.model small
.data
    dec_num             db      '1234'
    hex_num             dw      0
    mult_factor   dw    3e8h
    digit_count   dw    4
.code
```

```
    mov ax,@data;Initialize data segment
    mov ds,ax
    mov bx,0ah          ;Initialize division factor
    mov cx,digit_count  ;Digit counter
    mov si,offset dec_num ;Initialize memory pointer
up :
    mov al,[si]         ;read digit of BCD number
    and ax,000fh        ;mask required digit
    mul mult_factor     ;multiply by multiplication
                        factor
    add hex_num,ax      ;add to hex_num
    mov ax,mult_factor  ;change multiplication factor
    mov dx,00
    div bx
    mov mult_factor,ax
    inc si              ;increment memory pointer
    loop up             ;Is a last digit if no jump to up
    ends
    end
```

In above program, the BCD number 1234 is converted to 04D2. The conversion is started from 1st digit of BCD number i.e. 1, then 2, 3 so on. Hence, first conversion is started with the multiplication of 1 with 03E8H i.e. 1 X 03E8H, then 2 X 64h, 3 X 0Ah and at last 4 X 1.

The result of the above program after the execution is **04D2H** stored at **hex_num** variable.

In Alternative method, the conversion can be started from the last digit i.e. 4 X1, then 3 X 0Ah, 2 X 64H and 1 X 3E8H respectively. The following program performs this operation of conversion.

Program 34

```
    .model small
.data
    dec_num     db    '1234'   ;BCD Number
    hex_num     dw    0
    mult_factor dw    1
    digit_count dw    4
.code
    mov ax,@data;Initialize data segment
    mov ds,ax

    mov bx,0ah          ;Initialize division factor
    mov cx,digit_count  ;Digit counter
    lea si,dec_num+3    ;Initialize memory pointer
up :
    mov al,[si]         ;read last digit of BCD number
    and ax,000fh        ;mask required digit
    mul mult_factor     ;multiply by multiplication
                        ; factor add hex_num,ax
    mov ax,mult_factor  ;change multiplication factor
    mul bx
    mov mult_factor,ax
    dec si              ;increment memory pointer
    loop up             ;Is a last digit if no jump to up
    ends
    end
```

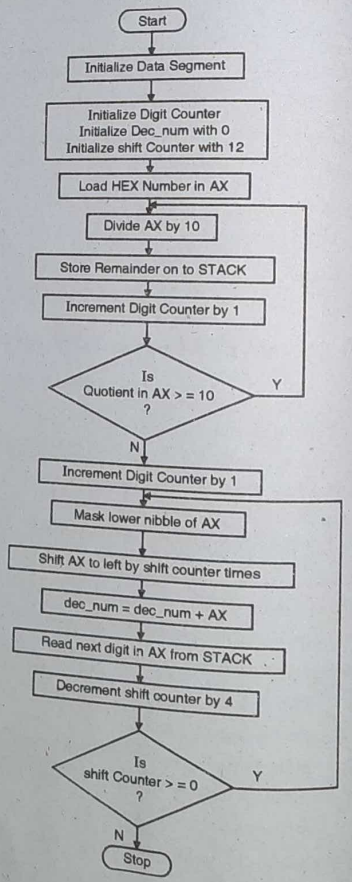
4.3.21 Convert Hexadecimal Number to BCD

- This conversion operation involves the previous steps, instead of multiplying, continue dividing the binary number [Hexadecimal] by 10 (0AH) until quotient is less than 10.
- The remainders, which can be only 0 through 9, successfully generate the BCD number.
- As an example, let's convert 4D2H back to BCD format as given below :

Divide by 10	Quotient	Remainder
4D2H / 0AH	7B	4
7BH / 0AH	0C	3
0CH / 0AH	01	2

- Since, the quotient i.e. 1 is less than the divisor (0AH), the operation is complete.
- The remainder along with the last quotient, form the BCD result, from right to left i.e. 1234.
- The following program performs above conversion operation.

Flowchart : Refer Flowchart 54.



Flowchart 54

Program 35

```
.model small
.data
    hex_num  dw  4D2h
    dec_num  dw  0
.code
    mov ax,@data;Initialize data segment
    mov ds,ax

    mov ch,00h          ;Digit counter
    mov bx,0ah          ;division factor
    mov ax,hex_num      ;load hex number
rpt :
    mov dx,00
    div bx              ;divide it by 10
    push dx             ;store remainder on stack
    inc ch              ;increment digit counter
    cmp ax,0ah          ;compare quotient with 10
    jge rpt             ;if quotient > 10 then repeat
                        ;division operation
    inc ch              ;increment digit counter at last
    mov cl,12           ;initialize shift counter
rpt3:
    and ax,0fh          ;mask lower nibble
    shl ax,cl           ;shift digit to appropriate
                        position
    add dec_num,ax      ;add with dec_num
    sub cl,4            ;decrement shift counter
    pop ax              ;read next BCD digit
    dec ch              ;decrement digit counter
    jnz rpt3            ;Is digit ≠ last digit then repeat
                        ;same shift and add operation
    mov ah,4ch          ;Terminate program & Exit to DOS
    int 21h
    ends
    end
```

- After the execution of the above program, we will get the result 1234 in memory variable dec_num which BCD equivalent of 4D2H.
- In above program, the value of digit counter indicates number of digits of converted BCD number.

4.3.22 BCD to ASCII Conversion

- If we see the ASCII code of all BCD numbers i.e. from 0 to 9, are from 30H to 39H.
- To convert any BCD number to ASCII, simply add 30H to the BCD number as the difference between BCD number and its ASCII is 30H
- To convert any ASCII to BCD, simply subtract 30H from the BCD number as the difference between BCD number and its ASCII is 30H

Algorithm

Step 1 : Initialize data segment
Step 2 : Initialize memory pointers for source and destination
Step 3 : Read BCD number from source memory
Step 4 : Convert to ASCII
Step 5 : Store into destination memory
Step 6 : Increment memory pointers by 1
Step 7 : Decrement byte counter by 1
Step 8 : If byte counter ≠ 0 then go to step 3
Step 9 : Stop

Program

```
    .model small
    .data
        bcd_no   db   1,2,3,4,5,6,7,8,9,0
        asc_no   db   10 dup(0)
    .code
    mov ax,@data;Initialize data segment
    mov ds, ax
    mov cx,10              ;Initialize byte counter
    mov si,offset bcd_no   ; Initialize memory pointers
    mov di,offset asc_no
up: mov al,[si]            ;Read BCD number
    add al,30H             ;Convert to ASCII
    mov [di],al            ;Store to memory
    inc si                 ;Increment memory pointers
    inc di
    loop up                ;Until all conversion
    ends
    end
```

4.3.23 ASCII to BCD Conversion

- To convert any ASCII to BCD, simply subtract 30H from the BCD number, simply replace ADD AL,30H with SUB AL,30H

Algorithm

Step 1 : Initialize data segment
Step 2 : Initialize memory pointers for source and destination
Step 3 : Read ASCII number from source memory
Step 4 : Convert to BCD
Step 5 : Store into destination memory
Step 6 : Increment memory pointers by 1
Step 7 : Decrement byte counter by 1
Step 8 : If byte counter ≠ 0 then go to step 3
Step 9 : Stop

Program

```
.model small
.data
        asc_no    db    '1','2','3','4','5','6','7','8','9','0'
        bcd_no    db    10 dup(0)
.code
        mov ax,@data      ;Initialize data segment
        mov ds, ax
        mov cx,10         ;Initialize byte counter
        mov si,offset asc_no ; Initialize memory pointers
        mov di,offset bcd_no
up:     mov al,[si]       ;Read ASCII number
        sub al,30H ;Convert to BCD
        mov [di],al ;Store to memory
        inc si            ;Increment memory pointers
        inc di
        loop up           ;Until all conversion
        ends
        end
```

Syllabus Topic : Count Numbers of 1 and 0 in 16 bit Number

4.3.24 Count Numbers of One's and Zero's in 8 Bit or 16 Bit Number

➔ **(MSBTE - S-14, S-15, W-16, W-17, S-18)**

Q. 4.3.45 Write an ALP to count number of '0' in AL register. (Ref. sec. 4.3.24) S-14, 4 Marks

Q. 4.3.46 Write an ALP to count the number of '1' in a 16 bit number. Assume the number to be stored in BX register. Store the result in CX register. (Ref. sec. 4.3.24) S-15, S-18, 4 Marks

Q. 4.3.47 Write an ALP to count number of 1's in register DL. (Ref. sec. 4.3.24) W-16, 4 Marks

Q. 4.3.48 Write ALP to count number of '0' in 16 bit number stored in AX register. (Ref. sec. 4.3.24) W-17, 4 Marks

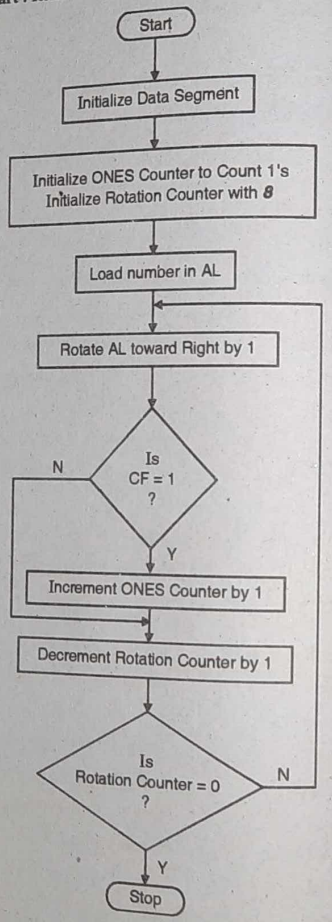
- The total numbers of 1's or 0's can count in any number by rotating that number toward right or left by either 8 times for 8 bit number or 16 times for 16 bit number.
- ROR or RCR or RCL or ROL instruction can be used to rotate any number to check how many ones or zeros are in the numbers.
- When we rotate number once to left or right, corresponding bit i.e. D_0 or D_7 initially goes to carry flag, then we can check carry flag by using JNC or JC to count numbers of ones or zeros.

➢ **Program 36 (a) : Count numbers of 1's in 8 bit number.**

➔ **(MSBTE - W-15)**

Q. 4.3.49 Write an ALP to count the number of '1' in a number stored in accumulator. (Ref. Program 36(a)) W-15, 4 Marks

Flowchart : Refer Flowchart 55(a).



Flowchart 55(a)

Algorithm

1. Initialize data segment.
2. Initialize rotation counter by 8.
3. Initialize ones counter to count number of 1's.
4. Load number.
5. Rotate number left or right by 1.

6. If CF ≠ 1 then go to step 8.
7. Increment ones counter by 1.
8. Decrement rotation counter by 1.
9. If rotation counter ≠ 0 then go to step 5.
10. Stop.

Program 36(a)

```
.model small
.data
    num    db  0ffh
    ones   db  0
.code
    mov ax,@data;Initialize data segment
    mov ds,ax
    mov cx,8       ;initialize rotation counter
    mov al,num     ;load number in AL
up : ror al,1      ;Rotate number by 1 bit right
    jnc dn         ;if bit ≠ 1 then go to dn
    inc ones       ;else increment ones by one
dn: loop up        ;decrement rotation counter
    ends           ;if rotation counter ≠ 0
                   ;then go to up
    end            ;stop
```

- After the execution of the above program, the result of the program will be 8 numbers of ones in memory variable **ones.**

➢ **Program 36(b) : Count numbers of 1's in 16 bit number.**

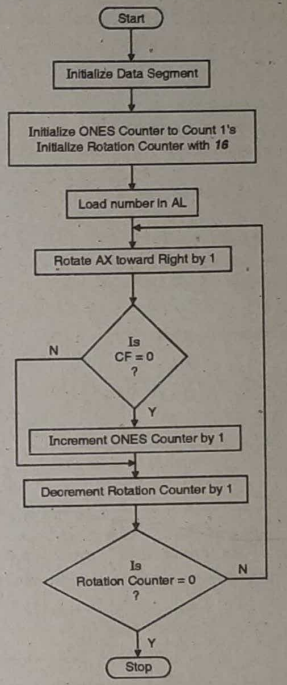
➔ **(MSBTE - S-16)**

Q. 4.3.50 Write an assembly language program to count numbers of '1' (ones) in 16 bit number stores in BX register. (Ref. Program 36(b)) S-16, 4 Marks

Algorithm

1. Initialize data segment.
2. Initialize rotation counter by 16.
3. Initialize ones counter to count number of 1's.
4. Load number.
5. Rotate number left or right by 1.
6. If CF ≠ 1 then go to step 8.
7. Increment ones counter by 1.
8. Decrement rotation counter by 1.
9. If rotation counter ≠ 0 then go to step 5.
10. Stop.

Flowchart : Refer Flowchart 55(b).



Flowchart 55(b)

➢ **Program 36(b)**

```
.model small
.data
    num    dw  0ffaah
    ones   db  0
.code
    mov ax,@data       ;Initialize data segment
    mov ds,ax
    mov cx,16          ;initialize rotation counter
    mov ax,num         ;load number in AX
up : ror ax,1          ;Rotate number by 1 bit right
    jnc dn             ;if bit ≠ 1 then go to dn
    inc ones           ;else increment ones by one
dn : loop up           ;decrement rotation counter
                       ;if rotation counter ≠ 0 then
                       ;go to up
    ends
    end                ;stop
```

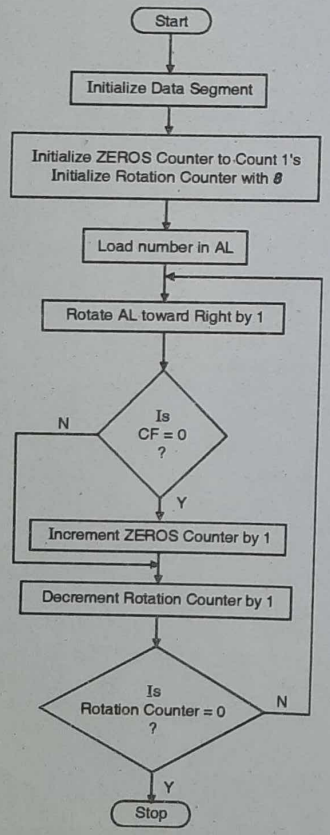
- After the execution of the above program, the result of the program will be 0CH (12 in BCD) numbers of ones in memory variable **ones.**

➢ Program 36 (c) : Count numbers of 0's in 8 bit number

Algorithm

1. Initialize data segment.
2. Initialize rotation counter by 8.
3. Initialize zeros counter to count number of 0's.
4. Load number.
5. Rotate number left or right by 1.
6. If CF ≠ 0 then go to step 8.
7. Increment zeros counter by 1.
8. Decrement rotation counter by 1.
9. If rotation counter ≠ 0 then go to step 5.
10. Stop.

Flowchart : Refer Flowchart 55(c).



Flowchart 55(c)

➢ Program 36(c)

```
.model small
.data
    num    db  0aah
    zeros  db  0
.code
    mov ax,@data        ;Initialize data segment
    mov ds,ax
    mov cx,8            ;initialize rotation counter
    mov al,num          ;load number in AL
up : ror al,1           ;Rotate number by 1 bit right
    jc dn               ;if bit ≠ 0 then go to dn
    inc zeros           ;else increment zeros by one
dn : loop up            ;decrement rotation counter
    ends                ;if rotation counter ≠ 0
                        then go to up
    end                 ;stop
```

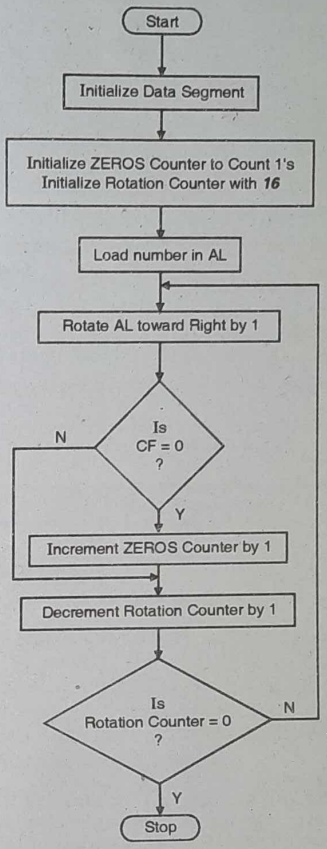
– After the execution of the above program, the result of the program will be 4 numbers of zeros in memory variable **zeros.**

➢ Program 36(d) : Count numbers of 0's and 1's in 16 bit number

Algorithm

1. Initialize data segment.
2. Initialize rotation counter by 16.
3. Initialize zeros counter to count numbers of 0's.
4. Initialize ones counter to count numbers of 1's.
5. Load number.
6. Rotate number left or right by 1.
7. If CF ≠ 0 then go to step 10.
8. Increment zeros counter by 1.
9. Go to step 11.
10. Increment ones counter by 1.
11. Decrement rotation counter by 1.
12. If rotation counter ≠ 0 then go to step 5.
13. Stop.

Flowchart : Refer Flowchart 55(d).



Flowchart 55(d)

➢ Program 36(d)

```
.model small
.data
    num    dw   0ffaah
    zeros  db   0
    ones   db   0
.code
    mov ax,@data        ;Initialize data segment
    mov ds,ax
    mov cx,16           ;initialize rotation counter
    mov ax,num          ;load number in AX
up :  ror ax,1          ;Rotate number by 1 bit right
    jc dn               ;if bit ≠ 0 then go to dn
    inc zeros
    jmp  next           ;else increment zeros by one
dn: inc ones
next: loop up           ;decrement rotation counter
    ends                ;if rotation counter ≠ 0 then to
                        up
    end                 ;stop
```

– After the execution of the above program, the result of the program will be 4 numbers of zeros in memory variable **zeros** and 0CH (12 in BCD) numbers of ones in memory variable **ones.**

➢ Program 37

Q. 4.3.51 Write a program to multiply the contents of AX by six using shift instruction. (Ref. Program 37)

Program

```
.model small
.data
    num1    dw   0002h
    num2    db   6
    result  dw   0
.code
    mov ax,@data        ; Initialize data segment
    mov ds,ax
    mov ax,num1         ; load multiplicand in ax
    mov dl,num2         ; load multiplier in dl
    mov bx,0            ; shift and add method
    mov cx,8            ; Shifting counter
up: add bx,ax
    shl dl,1
    jnc dn
    add bx,ax
dn: loop up
    mov result,bx       ; Store result
    mov ah,4ch          ; exit to DOS
    int 21h
    ends
    end
```

➢ Program 38

➔ **(MSBTE - S-14, S-17)**

Q. 4.3.52 Write an ALP to count number of '0' in AL register. (Ref.Program 38) **S-14, 4 Marks**

Q. 4.3.53 Write an ALP to count no. of zero's in BL register. (Ref. Program 38) **S-17, 4 Marks**

Program

```
.model small
.data
    num   db   0aah
    zeros db   0
.code
    mov ax,@data          ;Initialize data segment
    mov ds,ax
    mov cx,8              ;initialize rotation counter
    mov al,num            ;load number in AL
up : ror al,1             ;Rotate number by 1 bit right
    jc dn                 ;if bit ≠ 0 then go to dn
    inc zeros             ;else increment zeros by one
dn : loop up              ;decrement rotation counter
    ends                  ;if rotation counter ≠ 0 then go to up

    end                   ;stop
```

Chapter Ends...

❑❑❑

CHAPTER 5

UNIT - V

Procedure and Macro in Assembly Language Program

Syllabus

Procedure : Defining and calling, Procedure - PROC, ENDP, FAR, and NEAR Directives; CALL and RET instructions; Parameter passing methods, Assembly Language Programs using Procedure.

Macro : Defining Macros, MACRO and ENDM Directives. Macro with parameters. Assembly Language Programs using Macros.

Syllabus Topic : Procedures

5.1 Procedures

→ **(MSBTE - S-14, W-16, S-18)**

Q. 5.1.1 State the steps involved in ALP using procedure. (Ref. sec. 5.1) **S-14, 2 Marks**

Q. 5.1.2 What is Procedure? What are the two advantages of using procedure in our program. (Ref. sec. 5.1) **W-16, 4 Marks**

Q. 5.1.3 What do you mean by procedure? (Ref. sec. 5.1) **S-18, 1 Mark**

- A large program is difficult to implement even if an algorithm is available, hence it should be split into number of the independent tasks which can be easily designed and implemented.
- The process of splitting a large program into small tasks and designing them independently is known as **modular programming.**
- Large program are more prone to errors and it is difficult to locate and isolate errors.
- A repeated group of instruction in a program can be organized as subprogram. The subprograms are called as subroutine or procedures in assembly language programming which allows reuse of program code.
- A procedure is a set of the program statements that can be processed independently and reuse again and again.
- Here are the four steps that need to be accomplished in order to call and return from a procedure.
  1. Save return address
  2. Procedure call
  3. Execute procedure
  4. Return

Advantages

1. Simple modular programming.
2. Reduced work load and development time.
3. Debugging of the program and procedure are easier.
4. Reduction in the size of the main program.
5. Reuse of the procedure many times in the same program or in another program.
6. Library of the procedures can be implemented by combining well designed, tested and proven procedures.

Disadvantages

1. Extra code is required to integrate procedure i.e. CALL and RET instructions.
2. Required extra overhead while establishing a linkage between the calling program and the called procedure or vice versa, hence execution time is more.
3. As the run time overhead required for communication between small procedures is more than the execution time of the procedure and hence the use of small procedures are not preferable, instead, macro can be used.

Syllabus Topic : Defining and Calling, Procedure - PROC, ENDP, FAR and NEAR Directives

5.2 Defining Near or Far Procedures

→ **(MSBTE - S-14, W-14, S-15, W-15, S-16, W-16, S-17, W-17, S-18)**

Q. 5.2.1 Differentiate between re-entrant and recursive procedures.(Any two points) (Ref. sec. 5.2) **S-14, S-17, 4 Marks**

Q. 5.2.2 What do you meant by re-entrant procedure ? (Ref. sec. 5.2) W-14, W-15, 1 Mark

Q. 5.2.3 What do you mean by recursive procedure ? (Ref. sec. 5.2) W-14, 1 Mark

Q. 5.2.4 Describe reentrant procedure with the help of schematic diagram. (Ref. sec. 5.2) S-15, S-16, W-17, 4 Marks

Q. 5.2.5 What is recursive and re-entrant procedure. (Ref. sec. 5.2) W-16, 4 Marks

Q. 5.2.6 Explain re-entrant and recursive procedure. (Ref. sec. 5.2) S-18, 3 Marks

- The assembler directives PROC and ENDP are used to define a procedure.
- The directive PROC indicates the beginning of the procedure and the directive ENDP indicates the end of the procedure to the assembler.
- The directive PROC and ENDP must enclose the procedure code which defines the subroutine.
- The procedures must be defined within the code segment only.
- 8086 microprocessor has single interrupt signal used by some external device to interrupt the normal execution of program and call a specific procedure to Service the interrupt.
- Suppose 8086 was in the middle of executing factorial procedure when the interrupt signal occurred and also need to use the factorial procedure in the ISR.
- When interrupt occurs, execution goes to the ISR and ISR then calls the factorial procedure when it needs it.
- The RET instruction at the end of the factorial procedure returns execution to ISR.
- A special return instruction at the end of the ISR returns the execution to the factorial procedure where it was executing when the interrupt occurred as shown in Fig. 5.2.1.
- The factorial procedure must be written in such a way that it can be interrupted, used and re-entered without losing or writing over anything and such procedure is called as ***re-entrant***.
- To be ***re-entrant***, a procedure must first of all push the flags all registers on to stack used in the procedure and a program should use only registers or the stack to pass parameters.

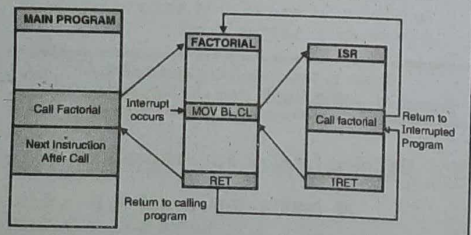


Fig. 5.2.1 : Re-entrant procedure flow diagram

- Other procedures are recursive procedure.
- A ***recursive*** procedure is a procedure, which calls within itself and are used to work with complex data structures called as trees.
- A recursive procedure is one that calls itself.
- There are two kinds of recursion : direct and indirect. In direct recursion, the procedure calls itself and in indirect recursion, the first procedure calls a second procedure, which in turn calls the first procedure.
- Recursion could be observed in numerous mathematical algorithms. For example, consider the case of calculating the factorial of a number.

Sr. No.	Re-entrant Procedure	Recursive Procedure
1.	The procedure which can be interrupted, used and "reentered" without losing or writing over anything.	It is the procedure which calls itself.
2.	The flow of control could be interrupted by a hardware interrupt and transferred to an Interrupt Service Routine (ISR)	The flow of control could be caused by CALL instruction and transferred to user's procedure.

5.2.1 Directives for Procedure

➔ **(MSBTE - W-16, S-17, W-17)**

Q. 5.2.7 State the functions of following directives 1. PROC 2. ENDP (Ref. sec. 5.2.1) W-16, 2 Marks

Q. 5.2.8 Give the syntax for defining a procedure. (Ref. sec. 5.2.1) S-17, 2 Marks

Q. 5.2.9 List directives used for procedure. (Ref. sec. 5.2.1) W-17, 2 Marks

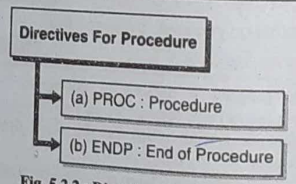


Fig. 5.2.2 : Directives for Procedure

➔ (a) PROC : Procedure

- The directive PROC indicates beginning of a procedure and follows with the name of the procedure.
- The term FAR or NEAR follows the PROC directive indicating the type of a procedure.
- If the term is not specified, then assembler assumes NEAR as the type specifier.
- The use of a procedure type specifier helps the assembler to decide whether to code RET as near return or far return.
- The directive PROC is used with the directive ENDP to enclose the procedure code.

General form

Procedure_Name PROC [NEAR/FAR]

Example

```
ADD PROC        NEAR
.....
.....
Procedure Codes
.....
.....
ADD ENDP
```

- This procedure can be called by using CALL instruction of 80x86 microprocessor whenever required such as CALL ADD.
- From this example, it is clear that the procedure will save a great amount of effort and time by avoiding the overhead of writing the repeated pattern of code.

➔ (b) ENDP : End of Procedure

- The directive ENDP informs the assembler the end of a procedure.
- The directive ENDP and PROC must enclose the procedure code.

General form

Procedure_Name ENDP

Examples

```
FACTORIAL ENDP ;    End of the procedure FACTORIAL
HEXTOASC  ENDP
```

Syllabus Topic : CALL and RET Instructions

5.2.2 Procedure Call [CALL Instruction]

➔ **(MSBTE - S-14, W-14, S-15, W-15, S-16, S-17, W-17, S-18)**

Q. 5.2.10 Differentiate between NEAR and FAR Calls. (Ref. sec. 5.2.2) S-14, S-18, 4 Marks

Q. 5.2.11 Write any two differences of FAR and NEAR procedure. (Ref. sec. 5.2.2) W-14, S-16, 2 Marks

Q. 5.2.12 Explain NEAR CALL and FAR CALL procedure. (Ref. sec. 5.2.2) S-15, W-15, 4 Marks

Q. 5.2.13 Explain NEAR and FAR procedure. (Ref. sec. 5.2.2) S-17, 4 Marks

Q. 5.2.14 Compare FAR and NEAR procedure. (Ref. sec. 5.2.2) W-17, 4 Marks

Q. 5.2.15 Explain CALL instruction. (Ref. sec. 5.2.2) W-17, S-18, 4 Marks

- The CALL instruction is used to transfer program control to a subprogram or a procedure by storing the return address on the stack.
- The call can be of two types
  - (a) Inter-Segment or near call
  - (b) Intra-Segment or far call
- A near call refers to a procedure call which is in the same code segment as the CALL instruction and a far call refers to a procedure call which is in the different code segment from that of the CALL instruction.

Syntax

CALL procedure_name

Operation performed

1. If NEAR CALL, then SP ← SP – 2
   Save IP on Stack
   IP← address of procedure
2. If FAR CALL, then SP ←SP – 2
   Save CS on stack
   CS ← New segment base address of the called procedure
   SP ← SP – 2
   Save IP on stack and
   IP ← New offset address of the called procedure

☞ Difference between NEAR Call and FAR Call

Sr. No.	Near Call	Far Call
1.	A *Near Call* refers to a procedure call which is in the same code segment as the CALL instruction.	A *Far Call* refers to a procedure call which is in the different code segment from that of the CALL instruction.
2.	Also called as Intra-segment call.	Also called as Inter-segment call
3.	A *Near call* replaces the old IP with new IP.	A *far call* replaces the old CS:IP pairs with new CS:IP pairs.
4.	The value of old IP is pushed on to the stack.	The value of the old CS:IP pairs are pushed on to the stack.
5.	Less stack locations are required.	More stack locations are required.

5.2.3 Procedure Return [RET Instruction]

➔ **(MSBTE - W-17, S-18)**

Q. 5.2.16 Describe RET instruction. (Ref. sec. 5.2.3) W-17, S-18, 4 Marks

- The instruction RET is used to transfer program control from the procedure back to the calling program i.e. main program or procedure following the CALL. The RET instruction are of two types :
  - (a) Near RET or inter segment return.
  - (b) Far RET or intra segment return.

- If a procedure is declared as near, the execution of the RET replaces the IP with a word from the top of the stack which contains the offset address of the instruction following the CALL instruction.
- Hence such return is called as near return because transfer of the control is within the segment.
- If a procedure is defined as far, the execution of RET instruction pops two words from the stack and places them into the registers IP and CS to transfer control to the calling program.

Syntax

RET

Operation performed

1. For NEAR Return, then IP ← content of top of stack
   SP ← SP + 2
2. For FAR Return, then
   IP ← contents of top of stack SP ← SP + 2
   CS ← contents of top of stack
   SP ← SP + 2

Syllabus Topic : Parameter Passing Methods

5.2.4 Parameter Passing in Procedure

➔ **(MSBTE - S-14, W-15)**

Q. 5.2.17 Explain various ways of parameter passing in 8086 assembly language procedure. (Ref. sec. 5.2.4) S-14, 8 Marks

Q. 5.2.18 Describe with suitable example how a parameter is passed on the stack in 8086 assembly language procedure. (Ref. sec 5.2.4) W-15, 4 Marks

Parameters can be passed between procedures in any of three ways : through general purpose registers, in an argument list, or on the stack.

5.2.4(A) Passing Parameters through the Registers

Q. 5.2.19 With one suitable example explain how a parameter is passed in register in 8086 assembly language procedure. (Ref. sec. 5.2.4(A))

- The processor does not save the state of the general-purpose registers on procedure calls.
- A calling procedure can thus pass up to six parameters to the called procedure by copying the parameters into any of these registers (except the SP and BP registers) prior to executing the CALL instruction.
- The called procedure can likewise pass parameters back to the calling procedure through general-purpose registers.

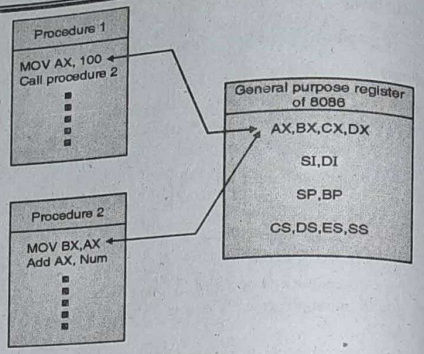


Fig. 5.2.3

5.2.4(B) Passing Parameters on the Stack

➔ **(MSBTE - S-15)**

Q. 5.2.20 Explain the stack operation. Why PUSH and POP instructions are used before and after CALL subroutine ? (Ref. sec. 5.2.4(B)) S-15, 4 Marks

- To pass a large number of parameters to the called procedure, the parameters can be placed on the stack, in the stack frame for the calling procedure.
- Here, it is useful to use the stack-frame base pointer (in the BP register) to make a frame boundary for easy access to the parameters.
- The stack can also be used to pass parameters back from the called procedure to the calling procedure.

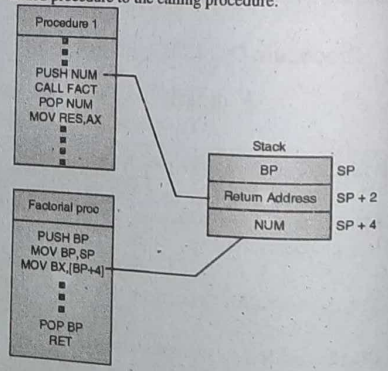


Fig. 5.2.4

Stack Operation

- Stack is a reserved memory location in a memory to store data temporarily.
- The operation of stack is LIFO (Last In First Out) during the data retrieval.

- The PUSH and POP instruction are available for stack operation in 8086.
- The PUSH instruction is used to write/store data on to the stack and the POP instruction is used to read data from the stack.
- When CALL instruction is used to execute subroutine, then some or all of the instruction of the subroutine may use the register for intermediate calculation which may contains some important data before calling subroutine.
- So during the execution of the subroutine, the old important data in the registers will be lost.
- Hence the PUSH instructions are used at the start of subroutine to store the contain of registers temporarily on the stack and at the end of subroutine POP instructions are used to restore same register with data from stack.

5.2.4(C) Passing Parameters in an Argument List

- An alternate method of passing a larger number of parameters (or a data structure) to the called procedure is to place the parameters in an argument list in one of the data segments in memory.

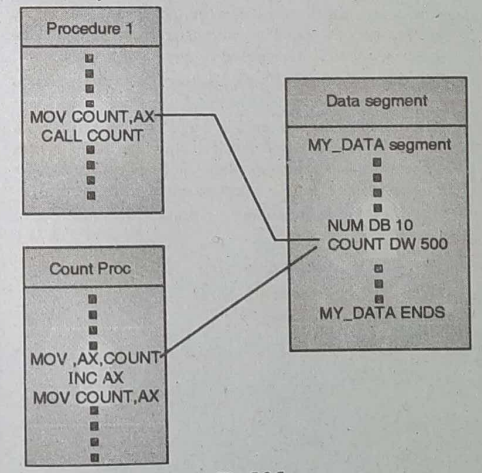


Fig. 5.2.5

- A pointer to the argument list can then be passed to the called procedure through a general purpose register or the stack.
- Parameters can also be passed back to the calling procedure in this same manner.

5.2.5 Saving Procedure State Information

- The processor does not save the contents of the general-purpose registers, segment registers or the FLAGS register on a procedure call.
- A calling procedure should explicitly save the values in any of the general-purpose registers that it will need when it resumes execution after a return.
- These values can be saved on the stack or in memory in one of the data segments.
- The PUSH and POP instruction facilitates saving and restoring the contents of the general purpose registers.
- If a called procedure changes the state of any of the segment registers explicitly, it should restore them to their former value before executing a return to the calling procedure.
- If a calling procedure needs to maintain the state of the FLAGS register it can save and restore all or part of the register using the PUSHF and POPF instructions.
- The PUSHF instruction pushes the word of the FLAGS register on the stack.
- The POPF instruction pops a word from the stack into the word of the FLAGS register.

Syllabus Topic : Macros

5.3 Macros

➔ **(MSBTE - S-14, W-15, S-16, W-16, S-17, S-18)**

Q. 5.3.1 What is MACRO? Define macros with an example. (Ref. sec. 5.3) S-14, S-16, S-18, 4 Marks

Q. 5.3.2 Define MACRO. List any four advantages of it. (Ref. sec. 5.3) W-15, 4 Marks

Q. 5.3.3 What is meant by macro's? Describe their uses. (Ref. sec. 5.3) W-16, 4 Marks

Q. 5.3.4 Define MACRO with its syntax. Also give two advantages of it. (Ref. sec. 5.3) S-17, 4 Marks

- In assembly language programs, small sequences of the codes of the same pattern are repeated frequently at different places which perform the same operation on the different data of the same data type.
- Such repeated code can be written separately as a macro.
- When assembler encounters a macro name later in the source code, the block of code associated with the macro name is substituted or expanded at the point of call, known as macro expansion.
- Hence macro is called as open subroutine.
- Macros should be used when its body has a few program statements; otherwise, the machine code of the program will be large on account of the same code being repeated in the position where macros are used.
- The process of defining macros and using them to simplify the programming process is known as macros programming.

Advantages

(a) Simplify and reduce the amount of repetitive coding.
(b) Reduces errors caused by repetitive coding.
(c) Makes program more readable.
(d) Execution time is less as compare to procedure as no extra instructions are required.

Syllabus Topic : Defining Macros

5.4 Defining Macros

→ **(MSBTE - W-14, S-15, S-17, W-17)**

Q. 5.4.1 Define MACRO. Write assembler directive used in MACRO. Explain. (Ref. secs. 5.4 and 5.4.1) **W-14, 4 Marks**

Q. 5.4.2 Give an example indicating how a MACRO can be defined and use of procedure & macro in 8086 ALP. (Ref. sec. 5.4) **S-15, 6 Marks**

Q. 5.4.3 Explain the directives used for defining MACRO. Give an example. (Ref. sec. 5.4) **S-17, 4 Marks**

Q. 5.4.4 Describe MACRO with syntax. (Ref. sec. 5.4) **W-17, 4 Marks**

- For macros that you want to include in your program, you must first define them.
- A macro definition appears before any defined segment.
- The assembler directive MACRO indicates to the assembler the beginning of the macro and the directive ENDM indicates the end of the macro to the assembler.
- The directives MACRO and ENDM must enclose the definitions, declaration, or a small part of a code which have to be submitted while translating them to machine code by the assembler.
- Since the program contains a definition of the macro, the assembler substitutes the body of the definition, generating the instructions called as macro expansion.

Syllabus Topic : MACRO and ENDM Directives

5.4.1 Directives for Macros

→ **(MSBTE - W-14)**

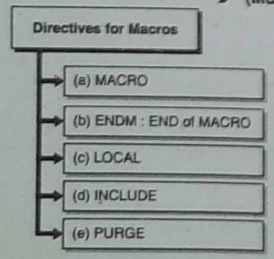


Fig. 5.4.1 : Directives for Macros

→ (a) MACRO

- The directive MACRO informs the assembler the beginning of a macro.
- It consists of name of a macro followed by keyword MACRO and macro arguments if any.
- The directives MACRO and ENDM must enclose the definition, declaration or a small part of code, which have to be substituted at the invocation of a macro.

General form

```
Macro_Name    MACRO
[Argument1,.....,ArgumentN]
```

Example

```
DISP  MACRO MSG
      PUSH AX
      PUSH DX
      MOV AH,09H
      LEA DX,MSG
      INT 21H
      POP DX
      POP AX
      ENDM
```

- The above macro whose name is DISP can be called by writing macro name along with its argument if any in the program wherever it is required i.e. number of times.
- The macro function are called as a open subroutine, macro gets expanded if a call is made to it.
- The difference between macros and procedures is that, a call to macro will be replaced with its body during assembly time whereas the call to the procedure will be an explicit transfer of program control to the called procedure during run time.
- The concept of macro is illustrated below.

```
.DATA
    MSG1 DB 'Well Com to Computer Department$'
    MSG2 DB 'Hardware Laboratory $'
.CODE
    DISP MSG1
    DISP MSG2
    .....
    ENDS
    END
```

From above example, it is clear that the macros will save a great amount of effort and time by avoiding the overhead of writing the repeated pattern of code.

→ (b) ENDM : END of MACRO

- The directive ENDM informs the assembler the end of the macro.
- The directive MACRO and ENDM must enclose the definition, declarations, or a small part of the code which have to be substituted at the invocation of the macro.

General form

```
ENDM
```


Example

```
DISP  MACRO MSG
      PUSH AX
      PUSH DX
      MOV AH,09H
      LEA DX,MSG
      INT 21H
      POP DX
      POP AX
      ENDM
```

→ (c) LOCAL

- Some macro requires that you define data item and instruction labels within the macro definition.
- If you use the macro more than once in the same program and the assembler defines the data item or label for each occurrence, the duplicate name would cause the assembler to generate error messages.
- To ensure that each generated name is unique, code the LOCAL directive immediately after the MACRO statement, even before comments.

General form

```
LOCAL dummy_1,dummy_2.....
                    ;One of more dummy argument.
```

Example

```
        ;Macro to Reverse the given string
        strrev macro str1,str2,count
            local up
            local next
            local exit
            push ax
            push si
            push di
            mov si,offset str1
next :      mov al,[si]
            cmp al,'$'
            je exit
            inc si
            inc count
            jmp next
exit :      mov di,offset str2
up1:        dec si
            mov al,[si]
            mov [di].al
            inc di
            dec count
            jnz up1
            mov al,'$'
            mov [di],al
            pop di
            pop si
            pop ax
            endm
```

- If you use above macro more than once in the same program, the duplicate label name i.e. up, exit and next will cause the assembler to generate an error message.
- So, this can be avoided by declaring labels i.e. up, next, exit as a local labels using LOCAL directives.

→ (d) INCLUDE

- The directive INCLUDE informs the assembler to include the statement defined in the include file.
- The name of the include file follows the statement INCLUDE.
- It is used to place all the data and frequently used macros into a file known as header or include file.
- The include file must exist in the directory specified in the path specification otherwise, and then assembler gives an error.
- The part of assembler which processes the include file is known as pre-processor.

General form

```
INCLUDE <file path specification with file name>
```

Example

```
INCLUDE C:\TASM\MACRO.LIB.
INCLUDE C:\TASM\BIN\MYPROC.LIB.
```

→ (e) PURGE

- Execution of an INCLUDE statement causes the assembler to include all the macro definitions that are in the specified library.
- Suppose, the library contains the macros DISP, READ_NUM, DISP_8BIT_NUM etc. but a program requires only READ_NUM.
- The PURGE directive enable you to "delete" the unwanted macros DISP and DISP_8BIT_NUM from the current assembly.

Examples

```
INCLUDE D:\TASM\MACRO.LIB.
PURGE DISP, DISP_8BIT_NUM.
```

Syllabus Topic : Macro with Parameters

5.4.2 Macro with Parameter or Arguments

- To make a macro flexible, you can define names in it as dummy arguments.
- In the macro explained with LABEL directive i.e. **strrev,** the macro argument **str1, str2 and count** will be substituted by the argument specified in the macro call.

Example

```
.model small
.data
```

```
str_s   db   'Computer$'
str_d   db   10 dup(0)
count   db   8
.code
        ::
        ::
        strrev str_s, str_d, count
        ::
        ::
        ends
        end
```

- In above example, the value of **str_s**, **str_d** and **count** will be substituted to macro argument **str1**, **str2** and **count** respectively.

5.4.3 Conditional MACRO Expansion

➔ **(MSBTE - S-15)**

Q. 5.4.5 What is conditional MACRO expansion ? Explain with one example. (Ref. sec. 5.4.3)

Q. 5.4.6 Write the difference between PROCEDURE and MACRO. (Ref. sec. 5.4.3) S-15, 4 Marks

- Most of the macro processors may also modify the sequence of statements generated for a macro expansion, depending on the arguments supplied in the macro invocation.
- Conditional Assembly is commonly used to describe this feature. It is also referred to as conditional macro expansion.
- Suppose we have written a macro to display string on the screen as given below

```
DISP MACRO MSG
MOV AH, 09H
MOV DX, OFFSET MSG
INT 21H
ENDM
```

- Assume, there are two string defined in STR1 and STR2, depending on the condition program may display either STR1 or STR2 as given below

```
.data
    Str1  db  'My name is TOM$'
    Str2  db  'My name is JERRY$'
    .code
        :
        :
        JNC NEXT
        DISP STR1
        JMP EXIT
    NEXT: DISP STR2
    EXIT: :
```

☞ **Difference between Procedure and Macro**

Sr. No.	Procedure	Macro
1.	A procedure is a set of the program statements that can be processed independently and reuse again and again.	A macro is a set of the program statements that can be reuse again and again by using macro name and hence macro is called as open subroutine.
2.	The machine code for the group of instruction in the procedures needs to be loaded in to main memory only once.	The machine code for the group of instruction in the MACRO needs to be loaded in to main memory wherever MACRO is used.
3.	Less main memory is required	More main memory is required
4.	CALL and RET instructions are used to call procedure which increase the overhead execution time involved in calling and returning from a procedures.	The macro avoids the overhead execution time involved in calling and returning from a macro as macro does not requires CALL and RET instructions.
5.	The procedures needs the stack compulsory.	The Macro does not need stack compulsory.

Syllabus Topic : Assembly Language Programs using Procedure

5.5 Programming using Procedures

5.5.1 Program to Perform Arithmetic Operation Such as Addition, Subtraction, Multiplication and Division using Procedures

➔ **(MSBTE - W-14, W-15)**

Q. 5.5.1 Write any assembly language program with re-entrant procedure. (Ref. sec. 5.5.1) W-14, W-15, 3 Marks

In this program we will write procedure for all arithmetic operations.

```
.model small
.data
        num1    dw 5432h
        num2    dw 99h
        res_add dw ?    ; result of addition
        res_sub dw ?    ; result of subtraction
        res_mul dd ?    ; result of multiplication
        res_quo dw ?    ; Quotient of division
        res_rem dw ?    ; Remainder of division
.code
        mov ax,@data    ;Initialize data segment
        mov ds,ax
call add_num        ;Call procedure for addition
call sub_num        ;Call procedure for subtraction
call mul_num        ;Call procedure for multiplication
call div_num        ;Call procedure for division
        mov ah,4ch      ;Exit to DOS
        int 21h
add_num proc        ; Procedure for addition
mov ax,num1
        add ax,num2
        mov res_add,ax
        ret
        endp
sub_num proc        ; Procedure for subtraction
        mov ax,num1
        sub ax,num2
        mov res_sub,ax
        ret
        endp
mul_num proc        ; Procedure for multiplication
mov ax,num1
        mul num2
        mov word ptr res_mul,ax
        mov word ptr res_mul+2,dx
        ret
        endp
div_num proc; Procedure for division
        mov ax,num1
        cwd
        div num2
        mov res_quo,ax
        mov res_rem,dx
        ret
        endp
        ends
end
```


5.5.2 Program to Arrange Numbers in the Array in Ascending Order using Procedure

```
.model small
.data
        array  dw 12h,11h,21h,9h,19h
.code
        mov ax,@data    ;Initialize data segment
        mov ds,ax
        call asc_order  ;call procedure to arrange
                        ;numbers in ascending order
        mov ah,4ch
        int 21h
asc_order  proc
        mov bx,5 ;Initialize pass counter
up1:
        lea si, array ;Initialize memory pointer
        mov cx,4      ;Initialize word counter
up:
        mov ax,[si]
        cmp al,[si+2]   ;Compare two number
        jc dn           ;if number < next number
                        ;then
        xchg ax,[si+2]  ;interchange numbers
        xchg ax,[si]
dn:     add si,2  ;increment memory pointer
        loop up   ;decrement word counter by 1 if
                  ;≠ 0 then up
        dec bx    ;decrement pass counter by 1 if
                  ;≠ 0 then up1
        jnz up1
        ret
        endp
        ends
        end
```

5.5.3 Program to Arrange Numbers in the Array in Descending Order using Procedure

Q. 5.5.2 What an ALP to arrange nos. In the arry in desending order using procedure. (Ref. sec. 5.5.3)

Program

```
        .model small
        .data
        array  dw    12h,11h,21h,9h,19h
        .code
        mov ax,@data        ;Initialize data segment
        mov ds,ax
        call dsc_order      ;call procedure to arrange
                            ;numbers in Descending order
        mov ah,4ch
        int 21h
        dsc_order  proc
        mov bx,5            ;Initialize pass counter
up1:
        lea si,array        ;Initialize memory pointer
        mov cx,4            ;Initialize word counter
up:
        mov ax,[si]
        cmp al,[si+2]       ;Compare two number
        jnc dn              ;if number > next number then
        xchg ax,[si+2]      ;interchange numbers
        xchg ax,[si]
dn :    add si,2            ;increment memory pointer
        loop up             ;decrement word counter if ≠ 0
                            ;then up
        dec bx              ;decrement pass counter if ≠ 0
                            ;then go to up1 jnz up1
        ret                 ;return to calling program
        endp
        ends
        end
```

5.5.4 Program to Find Smallest Number from the Array using Procedure

➔ **(MSBTE - S-17)**

Q. 5.5.3 Write an ALP to find smallest number using procedure. (Ref. sec. 5.5.4) **S-17, 4 Marks**

```
        .model small
        .data
        array  dw    134h,65h,876h,976h,2h
        small  dw    0
        .code
        mov ax,@data    ;Initialize data segment
        mov ds,ax
        call smallest   ;call procedure to find smallest
                        ; number
        mov small,ax
        mov ah,4ch      ;Exit to DOS
        int 21h
        smallest proc
        mov cx,5        ;Initialize word counter
        mov si,offset array ;Initialize memory pointer
        mov ax,[si]     ;read number from array
        dec cx
up :    inc si
        inc si
        cmp ax,[si]     ;compare it with next number
        jc next         ;if number is smallest then
                        compare it
        mov ax,[si]     ;it next number
next:                   ;decrement word counter by 1
        loop up         ;if it is NOT ZERO, compare with
                        next number in array
        ret             ;return to calling program
        endp
        ends
        end
```

5.5.5 Program to Find Largest Number from the Array using Procedure

```
        .model small
        .data
              array  dw    134h,65h,876h,976h,2h
              large  dw    0

        .code
        mov ax,@data        ;Initialize data segment
        mov ds,ax
        call largest        ;call procedure to find
                            smallest number
        mov large,ax
        mov ah,4ch          ;Exit to DOS
        int 21h
largest proc
        mov cx,5            ;Initialize word counter
        mov si,offset array ;Initialize memory pointer
        mov ax,[si]         ;read number from array
        dec.cx
up :    inc si
        inc si
        cmp ax,[si]         ;compare it with next number
        jnc next            ;if number is smallest then compare
        mov ax,[si]
next : loop up              ;decrement word counter if NOT
                            ; ZERO, compare with next
                            ; number in array
        ret                 ;return to calling program
        endp
        ends
        end
```

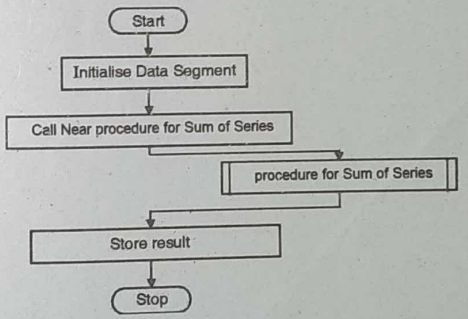

5.5.6 Program for Addition of Series of 8 bit Numbers using Procedure

➔ **(MSBTE - S-16, W-17)**

Q. 5.5.4 Write an assembly language program for sum of series of 10 numbers using procedure. *(Ref. sec. 5.5.6)* **(S-16, 4 Marks)**

Q. 5.5.5 Write ALP for sum of series of 10 numbers using procedure. Also draw flow chart for the same. *(Ref. sec. 5.5.6)* **(W-17, 8 Marks)**

Flowchart



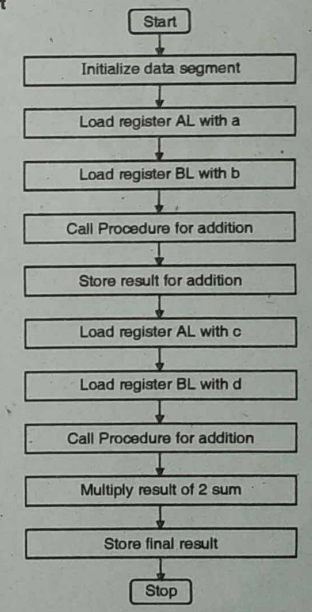
Program

```
.model small
.data
        list    db   1,2,3,4,5,6,7,8,9,10
        res     db   0
        count   db   10
.code
        mov ax,@data    ;Initialize data segment
        mov ds,ax
        call sum        ;call procedure to sum of series
sum     proc
        mov cl,count    ; initialize byte counter
        mov si,offset list ; initialize memory pointer
        mov al,0        ; initialize sum variable
up:     add al,[si]     ; add numbers from series with
                          sum
        inc si          ; increment memory pointer
        dec cl          ; decrement byte counter
        jnz up          ; if byte counter is not zero
                        ; then go to up
        ret             ; return to calling program
        endp
        ends
```

5.5.7 Program using Procedure for Performing the Operations Z = (A + B) * (C + D)

Q. 5.5.6 Write assembly program using procedure for performing the operations Z = (A + B) * (C + D). Draw flow chart and write Result. (Ref. sec. 5.5.7)

Flowchart



Program

```
.model small
.data
        a       db   1
        b       db   2
        c       db   3
        d       db   4
        z       dw   ?
        asum    db   ?
.code
        mov ax,@data    ; Initialize data segment
        mov ds,ax
        mov al,a        ; load al with a
        mov bl,b        ; load bl with b
        call add_byte   ;call procedure for addition
        mov asum,al     ; store result of addition
        mov al,c        ; load al with c
        mov bl,d        ; load bl with d
        call add_byte   ;call procedure for addition
        mul asum        ;multiply the result of 2 sum
        mov z,ax        ; store final result
add_byte    proc
        add al,bl
        ret
        endp
        ends
        end
```

5.5.8 Procedure to find Factorial of a Number

➔ **(MSBTE - W-14, S-17, S-18)**

Q. 5.5.7 Write an assembly language program using recursive procedure to find factorial of a number. (Ref. sec. 5.5.8) **W-14, 3 Marks**

Q. 5.5.8 Write a procedure to find factorial of a number. (Ref. sec. 5.5.8) **S-17, S-18, 4 Marks**

```
fact    proc
        push ax
        push bx
        push dx
        mov ax,num
        mov bx,ax
        dec bx
up:     mul bx
        mov fact_lsb,ax
        mov fact_msb,dx
        dec bx
        cmp bx,0
        jnz up
        pop dx
        pop bx
        pop ax
        ret
        endp
```

5.5.9 Program to find Factorial of a Number using Procedure

The program given below is used to find factorial of number 8 or less than 8.

```
.model small
.data
        num         dw   8
        fact_lsw    dw   0
        fact_msw    dw   0
.code
        mov ax,@data
        mov ds,ax
        call fact   ;Call procedure to find factorial of 8
fact  proc
        push ax
        push bx
        push dx
        mov ax,num
        mov bx,ax
        dec bx
up:  mul bx
        mov fact_lsw,ax
        mov fact_msw,dx
        dec bx
        cmp bx,0
        jnz up
        pop dx
        pop bx
        pop ax
        ret
        endp
        ends
        end
```


5.5.10 Program to Multiply 8 bit Numbers using NEAR Procedure

➔ **(MSBTE - W-14, W-15)**

Q. 5.5.9 Write an assembly language program to multiply two 8 bit numbers using NEAR procedure. Also draw the flowchart for the same. (Ref. sec. 5.5.10) **W-14, 8 Marks**

Q. 5.5.10 Write an ALP to multiply two 8 bit numbers using NEAR procedure. (Ref. sec. 5.5.10) **W-15, 4 Marks**

Flowchart

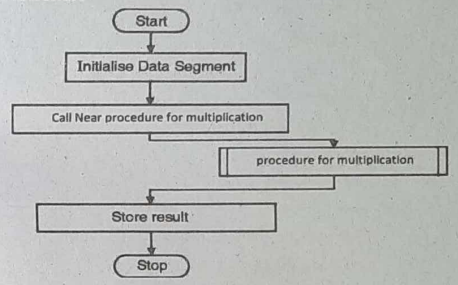


Fig. 5.5.1

Program

```
.model small
.data
        num1   db 99h
        num2   db 99h
        res_mul dw ?   ;result of multiplication
.code
        mov ax,@data   ;Initialize data segment
        mov ds,ax
        call mul_num   ;Call procedure for multiplication
        mov ah,4ch     ;Exit to DOS
        int 21h
mul_num proc           ;Procedure for multiplication
        mov al,num1
        mul num2
        mov res_mul,ax
        ret
        endp
        ends
        end
```

5.5.11 Program using Procedure to Add Two BCD Numbers

➔ **(MSBTE - W-17)**

Q. 5.5.11 Write ALP using procedure to add two BCD numbers. (Ref. sec. 5.5.11) **W-17**

In this program we will write procedure for all arithmetic operations.

```
.model small
.data
        num1    db   99Hh
        num2    db   99h
        res_lsb db   ?   ; result of addition
        res_msb db   0
.code
        mov ax,@data    ;Initialize data segment
        mov ds,ax
        call add_num    ;Call procedure for addition
        mov ah,4ch      ;Exit to DOS
        int 21h
add_num proc            ; Procedure for addition
        mov al,num1
        add al,num2
        daa
        jnc dn
        inc res_msb
dn:     mov res_lsb,al
        ret
        endp
```

Syllabus Topic : Assembly Language Programs using Macros

5.6 Programming using Macros

5.6.1 Simple Program for Addition of Two Numbers using Macro

```
        add_num
        .model small
        add_num     macro no1,no2,result
                    mov ax,no1
                    add ax,no2
                    mov result,ax
                    endm
        .data
                    num1    dw   1234h
                    num2    dw   4321h
```

```
        res     dw    ?
.code
        mov ax,@data
        mov ds,ax
        add_num num1,num2,res
        ends
        end
```

5.6.2 Smallest Number in the Array using Macro

```
.model small
small_w macro array,length,smallest
        local up
        local next
        push ax
        push cx
        push si
        mov cl,length
        mov si,offset array
        mov ax,[si]
        dec cl
up :    add si,2
        cmp ax,[si]
        jc next
        mov ax,[si]
next :
        dec cl
        jnz up
        mov smallest,ax
        pop si
        pop cx
        pop ax
        endm
.data
        list_1    dw    12h,19h,98h,54h,73h
        list_2    dw    65h,75h,85h,06h,45h
        count     db    5
        small_1   dw    ?
        small_2   dw    ?
.code
        mov ax,@data
        mov ds,ax
        small_w list_1,count,small_1
        small_w list_2,count.small_2
        ends
        end
```

5.6.3 Largest Number in the Array using Macro

```
.model small
large_w macro array,length,largest
        local up
        local next
        push ax
        push cx
        push si
        mov cl,length
        mov si,offset array
        mov ax,[si]
        dec cl
up :    add si,2
        cmp ax,[si]
        jc next
        mov ax,[si]
next:   dec cl
        jnz up
        mov largest,ax
        pop si
        pop cx
        pop ax
        endm
.data
        list_1    dw    12h,19h,98h,54h,73h
        list_2    dw    65h,75h,85h,06h,45h
        count     db    5
        large_1   dw    ?
        large_2   dw    ?
.code
        mov ax,@data
        mov ds,ax
        large_w list_1,count,large_1
        large_w list_2,count,large_2
        ends
        end
```

5.6.4 Program to Concatenating Source String to Destination String

```
.model small
show_msg macro msg
        push ax
        push dx
        mov ah,09h
        mov dx,offset msg
        int 21h
        pop dx
        pop ax
        endm
.data
        str_s   db  'COMPUTER ENGINEERING$'
        str_d   db  'INFORMATION TECHNOLOGY$'
        msg1    db  10,13,'The source String : $',10,13
msg2    db  10,13,      'The destination String : $',10,13
msg3    db  10,13       ,'After Concatenation.....: $',10,13
count   db  0
.code
        mov ax,@data    ;Initialize data segment
        mov ds,ax
        show_msg msg1 ;display messages
        show_msg str_s
        show_msg msg2
        show_msg str_d
        mov si,offset str_s  ; move memory pointer to last
next :  mov al,[si]          ; character of the source string
        cmp al,'$'
        je exit
        inc si
        inc count
        jmp next
exit :  mov di,offset str_d          ; copy character from
                                     ; destination string
up :    mov al,[di]                  ; to source string
        cmp al,'$'
        je exit1
        mov [si],al
        inc si
        inc di
        jmp up
        exit1: mov al,'$'
        mov [si],al
        show_msg msg3          ; display concatenated string
        show_msg str_s
        mov ah,4ch
        int 21h
        ends
            end
```

Output of above program

```
G:\tasm>strcatv.
The source string       : COMPUTER ENGINEERING.
The destination string  :_INFORMATION TECHNOLOGY.
After concatenation.... :
COMPUTER ENGINEERING_INFORMATION TECHNOLOGY.
G:\tasm>
```


5.6.5 Display String On the Screen

```
.model small
disp macro msg
        push ax
        push dx
        mov ah,09h
        mov dx,offset msg
        int 21h
        pop dx
        pop ax
        endm
.data
        str     db     'My name is VIJAY$'
.code
        mov ax,@data
        mov ds,ax
        disp str
        mov ah,4ch
        int 21h
        ends
        end
```

Output of above program

```
G:\tasm>dispv
My name is VIJAY
G:\tasm>
```

5.6.6 Using Macros, Assembly Language Program to Solve $P = x^2 + y^2$ where x and y are 8 bit Number

Q. 5.6.1 Using macros writ assembly language program to solve $P = x^2 + y^2$ where x and y are 8 bit number. (Ref. sec. 5.6.6)

Program

```
.model small
sqr_num     macro  no1, sqr
            mov al,no1
            mul no1
            mov sqr,ax
            endm
.data
            x       db     34h
            y       db     21h
            sx      dw     ?
            sy      dw     ?
            p       dw     ?
```

```
.code
        mov ax,@data    ;Initialize data segment
        mov ds,ax
        sqr_num x, sx   ; Find x^2
        sqr_num y, sy   ; Find y^2
        mov ax, sx
        add ax, sy      ; Perform x^2+y^2
        mov p, ax
        ends
        end
```

5.6.7 Using Macros, Assembly Language Program to Solve Z = (A + B) * (C + D) where A, B, C and D are 8 Bit Numbers

➔ **(MSBTE - S-16)**

Q. 5.6.2 Write an assembly language program using MACRO to perform following operations X = (A + B) * (C + D). (Ref. sec. 5.6.7) **S-16, 4 Marks**

Program

```
.model small

sum_num   macro no1,no2, res
          mov al, no1
          add al, no2
          mov res,al
          endm

.data
          a     db  1
          b     db  2
          c     db  3
          d     db  4
          r1    db      ?
          r2    db      ?
          z     dw  ?
          sum   db  ?

.code
          mov ax,@data    ;Initialize data segment
          mov ds,ax
          sum_num a,b,r1  ; Find (a+b)
          sum_num c,d,r2  ; Find (c+d)
          mov al, r1
          mul r2          ; Perform (a+b)*(c+d)
          mov z, ax
          ends
          end
```

Chapter Ends...

❑❑❑